Bellek Sınırlaması Altında Etkin Paralel Metin Erişimi için Seçici Veri Replikasyonu

**Proje No:** 109E019

Prof.Dr. Cevdet Aykanat  
Ata Türk  
Reha Oğuz Selvitopi  
Volkan Yazıcı

EYLÜL 2011  
ANKARA
Önsöz
Bu proje TÜBİTAK tarafından 109E019 proje numarası altında desteklenmiştir. Projede terim-tabanlı bölümlenmiş paralel sorgu işleme sistemlerinde hiperçizge modellerini baz alarak akıllı çoklama yapabilen teknikler geliştirilmiş ve bu teknikler gerçek zamanlı paralel sorgu işleme modellerinde test edilerek geçerlilikleri kanıtlanmıştır.
<table>
<thead>
<tr>
<th>İçindekiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablo ve Şekil Listeleri ................................................................. 4</td>
</tr>
<tr>
<td>Abstract ................................................................................ 6</td>
</tr>
<tr>
<td>Özet .................................................................................. 7</td>
</tr>
<tr>
<td>1 GİRİŞ .............................................................................. 8</td>
</tr>
<tr>
<td>1.1 Paralel Sorgu İşleme Sistemleri ....................................................... 8</td>
</tr>
<tr>
<td>1.1.1 Ters Dizin Yapısı .................................................................. 9</td>
</tr>
<tr>
<td>1.1.2 Sorgu İşleme .................................................................. 9</td>
</tr>
<tr>
<td>1.1.3 Paralel Sistemlerde Ters Dizin Organizasyonları .......... 9</td>
</tr>
<tr>
<td>1.1.4 Paralel Sorgu İşleme ......................................................... 9</td>
</tr>
<tr>
<td>1.1.5 Paralel Sistemlerde Ters Dizin Çöklemesi ............... 10</td>
</tr>
<tr>
<td>1.2 Hipercizge Bölümleme Problemi .................................................. 10</td>
</tr>
<tr>
<td>1.3 Katkılar ....................................................................... 11</td>
</tr>
<tr>
<td>2 LİTERATÜR ÖZETİ .................................................................. 12</td>
</tr>
<tr>
<td>3 TEK AŞAMALI FM-TABANLI ÇOKLAMALI HİPERÇIZGE BÖLÜMLEME .......... 14</td>
</tr>
<tr>
<td>3.1 Çoklamalı (Replicated) FM (rFM) .............................................. 15</td>
</tr>
<tr>
<td>3.1.1 Tanımlar ..................................................................... 15</td>
</tr>
<tr>
<td>3.1.2 Genel rFM Algoritması ..................................................... 16</td>
</tr>
<tr>
<td>3.1.3 Hipеркаляр Kritikliği ...................................................... 18</td>
</tr>
<tr>
<td>3.1.4 rFM Algoritmasının Detayları ........................................... 19</td>
</tr>
<tr>
<td>3.2 Çok Seviyeli Yöntem ve Özyinelemeli Bölümleme .................... 25</td>
</tr>
<tr>
<td>3.3 Kopya veya Bacak Seçimi ve Küme Kapama Sezgiseli .......... 26</td>
</tr>
<tr>
<td>3.4 Çoklama Miktarı Dağıtımı ...................................................... 28</td>
</tr>
<tr>
<td>3.5 Bölümlemeye Ait Deneysel Sonuçlar ...................................... 29</td>
</tr>
<tr>
<td>3.5.1 Deneysel Düzenek .......................................................... 29</td>
</tr>
<tr>
<td>3.5.2 Performans Değerlendirmeleri ............................................ 30</td>
</tr>
<tr>
<td>4 İKİ AŞAMALI DM-TABANLI ÇOKLAMALI HİPERÇIZGE BÖLÜMLEME ............ 35</td>
</tr>
<tr>
<td>4.1 İki Parçalı DM-Tabanlı Çözümleme ......................................... 35</td>
</tr>
<tr>
<td>4.2 K Parçalı Bir Hipercizge Bölümlemesi için Dengeyi Sağlayan Min-Kesit Çöklama Kümeleri ................. 39</td>
</tr>
<tr>
<td>4.3 Hk Hipercizgelerinin İnşası ................................................... 40</td>
</tr>
<tr>
<td>4.4 Hk Hipercizgeleri Üzerinde Düğüm Seçimi ............................. 43</td>
</tr>
<tr>
<td>4.5 Hk Hipercizgelerinin Kıcılıtülmesi ........................................ 44</td>
</tr>
</tbody>
</table>
4.6 Bölümleneye Ait Deneysel Sonuçlar ........................................................................ 46
  4.6.1 Veri Kümeleri ........................................................................................................ 47
  4.6.2 İlk Aşama: Hiperçizge Bölümleme ................................................................... 47
  4.6.3 İkinci aşama: Çoklama ......................................................................................... 47
  4.6.4 Sıkıştırma Algoritmalarının Karşılaştırılması .................................................... 48

5 GERÇEK ZAMANLI PARALEL SORGU İŞLEME SİSTEMİ ........................................ 49
  5.1 Bölümleme ................................................................................................................ 51
    5.1.1 Döküman Yiğını Özellikleri ve Sentetik Sorgu Üretimi .................................. 51
    5.1.2 Bölümleme Modeli ve Parametreleri ................................................................. 51
    5.1.3 Bölümleme Sonuçları ......................................................................................... 53
  5.2 Paralel Sorgu İşleme ................................................................................................. 54
    5.2.1 Paralel Sistem Mimarisi .................................................................................... 54
    5.2.2 İki Farklı Sorgu Şeması .................................................................................. 56
    5.2.3 Zamanlama (Scheduling) Algoritmaları ............................................................... 58
    5.2.4 Sistem Özellikleri ve Parametreleri ................................................................. 58
    5.2.5 Sorgu Performans Sonuçları ............................................................................ 59

6 SONUÇ .......................................................................................................................... 67

7 REFERANSLAR .............................................................................................................. 67
<table>
<thead>
<tr>
<th>Şekil</th>
<th>Açıklama</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Klasik bir arama motoru yapısı.</td>
</tr>
<tr>
<td>2</td>
<td>$v_1$ düğümünün taşınması ve çoklanması.</td>
</tr>
<tr>
<td>3</td>
<td>Çoklanmış $v_1$ düğümüne ait iki örneğin azlanması.</td>
</tr>
<tr>
<td>4</td>
<td>Örnek hiperçizgedeki düğümlerin ilk kazanç değerleri ile hiperkenarların bacak dağılımları.</td>
</tr>
<tr>
<td>5</td>
<td>$v_4$ düğümünün taşınmasından sonra kazanç değerleri ve bacak dağılımları.</td>
</tr>
<tr>
<td>6</td>
<td>$v_6$ düğümünün çoklanmasından sonra kazanç değerleri ve bacak dağılımları.</td>
</tr>
<tr>
<td>7</td>
<td>$v_7$'e ait $B$ bölümündeki örneğin azlanmasından sonra kazanç değerleri ve bacak dağılımları.</td>
</tr>
<tr>
<td>8</td>
<td>Bölümleme sonrası $n_j$ hiperkenarı için bacak seçimi. (a) Bacak seçiminden önceki ilk bölümleme. (b) Birinci seçim alternatifi, $n_j$'nin bağlı olduğu bölüm sayısı 3, (c) İkinci seçim alternatifi, $n_j$'nin bağlı olduğu bölüm sayısı 1.</td>
</tr>
<tr>
<td>9</td>
<td>(a) Seçimden önceki ilk bölümleme. (b) Bacak seçim algoritmasının ilk aşamasında yapılan seçimler. (c) Bacak seçim algoritmasının ikinci aşamasında yapılan seçimler.</td>
</tr>
<tr>
<td>10</td>
<td>Artan $K$ (10a) ve artan $p$ (10b) değerleri için PaToH+MF ve rpPaToH için paha değerleri.</td>
</tr>
<tr>
<td>11</td>
<td>Örnek bir bölümlemede V, ve V, arasındaki kesit.</td>
</tr>
<tr>
<td>12</td>
<td>Şekil 11'deki kesitler için oluşturulan iki kümelı çizge.</td>
</tr>
<tr>
<td>13</td>
<td>Şekil 12'de belirlenen düğümlerin çoklanması sonucu oluşan ikinci düğüm çoklanması safhası.</td>
</tr>
<tr>
<td>14</td>
<td>DM çözümlemesi kullanarak belli parçaların bulunmasının aşamaları.</td>
</tr>
<tr>
<td>15</td>
<td>3 parçadan oluşan örnek bir hiperçizge.</td>
</tr>
<tr>
<td>16</td>
<td>Örnek $H^\text{cut}$ ve $H^\text{con}$ hiperçizgeleri.</td>
</tr>
<tr>
<td>17</td>
<td>Örnek $H^\text{con}$ hiperçizgesinin DM ayrıımı ile küçültülmesi.</td>
</tr>
<tr>
<td>18</td>
<td>İki aşamalı paralel sorgu işleme sisteminin genel hatları. 1. aşamada sentetik bir sorgu dosyası kullanılarak bölümleme işlemi gerçekleştirilmiştir. Sonraki aşamada ise bu bölümleme bilgisi kullanılarak kölelerin ters liste dizinleri oluşturulmuştur ve oluşturulmuş sentetik sorgular kullanılarak paralel sorgu işleme sistemi çalıştırılmaktadır.</td>
</tr>
<tr>
<td>19</td>
<td>repl-ABCserver paralel sorgu işleme sisteminde örnek bir sorgunun terim-tabanlı bölümülenmiş sistemde işlenmesi.</td>
</tr>
<tr>
<td>20</td>
<td>Sorgu işleme sırasında kullanılan VE ve VEYA yöntemlerinin ters dizin sonuçları açısından farklılıklar.</td>
</tr>
<tr>
<td>21</td>
<td>Merkezi simscarın kendisine gelen kısımi cevap listelerini (KCL) iki farklı yöntemde birleştirmesi.</td>
</tr>
<tr>
<td>Tablo</td>
<td>Açıklama</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>Veri kümleri özellikleri.</td>
</tr>
<tr>
<td>2</td>
<td>bis+nor şemasına göre normalize edilmiş dört farklı rpPaToH şemasının bütün veri kümlerine ait paha değerleri ortalamaları.</td>
</tr>
<tr>
<td>3</td>
<td>PaToH ve rpPaToH için yük dengesizliği (%LI) ve çoklama kullanım (%RU) yüzdeleri.</td>
</tr>
<tr>
<td>4</td>
<td>PaToH+MF ve rpPaToH için paha değerleri (x 10³).</td>
</tr>
<tr>
<td>5</td>
<td>rpPaToH’un PaToH’a göre bütün kümlerindeki ortalama yüzde paha azalma değerleri.</td>
</tr>
<tr>
<td>6</td>
<td>rpPaToH’un PaToH’a göre normalize edilmiş bütün veri kümlerindeki ortalama koşma zamanı değerleri.</td>
</tr>
<tr>
<td>7</td>
<td>Sonuçlarda kullanılan terimler.</td>
</tr>
<tr>
<td>8</td>
<td>Veri kümesi özellikleri.</td>
</tr>
<tr>
<td>9</td>
<td>İlk aşamadaki hiperçizge bölümlene sonucunda elde edilen bölümlenelerin özellikleri.</td>
</tr>
<tr>
<td>10</td>
<td>Çoçlama sonuçları.</td>
</tr>
<tr>
<td>11</td>
<td>20.000 sorgu logundan üreten hiperçizge özellikleri. Bahsedildiği üzere her bir sorgu hiperkenar olarak modellendiğinden hiperkenar sayısı sorgu sayısına eşittir (Ort. = Ortalama).</td>
</tr>
<tr>
<td>12</td>
<td>Bölüm sayısı 4 (K = 4) için çeşitli değerler ve karşılaştırmaları. Gelişme oranları kesit değerleri üstünden olup, PaToH-CON ve PaToH-CUT değerleri baz alınarak hesaplanmıştır.</td>
</tr>
<tr>
<td>13</td>
<td>Bölüm sayısı 8 (K = 8) için çeşitli değerler ve karşılaştırmaları. Gelişme oranları kesit değerleri üstünden olup, PaToH-CON ve PaToH-CUT değerleri baz alınarak hesaplanmıştır.</td>
</tr>
<tr>
<td>14</td>
<td>Bölüm sayısı 16 (K = 16) için çeşitli değerler ve karşılaştırmaları. Gelişme oranları kesit değerleri üstünden olup, PaToH-CON ve PaToH-CUT değerleri baz alınarak hesaplanmıştır.</td>
</tr>
<tr>
<td>15</td>
<td>K = 4, Küme Örtme Alg, VE</td>
</tr>
<tr>
<td>16</td>
<td>K = 4, Küme Örtme Alg, VEYA</td>
</tr>
<tr>
<td>17</td>
<td>K = 8, Küme Örtme Alg, VE</td>
</tr>
<tr>
<td>18</td>
<td>K = 8, Küme Örtme Alg, VEYA</td>
</tr>
<tr>
<td>19</td>
<td>K = 16, Küme Örtme Alg, VE</td>
</tr>
<tr>
<td>20</td>
<td>K = 16, Küme Örtme Alg, VEYA</td>
</tr>
<tr>
<td>21</td>
<td>K = 4, Dinamik Yük Dengeleme Alg, VE</td>
</tr>
<tr>
<td>22</td>
<td>K = 4, Dinamik Yük Dengeleme Alg, VEYA</td>
</tr>
<tr>
<td>23</td>
<td>K = 8, Dinamik Yük Dengeleme Alg, VE</td>
</tr>
<tr>
<td>24</td>
<td>K = 8, Dinamik Yük Dengeleme Alg, VEYA</td>
</tr>
<tr>
<td>25</td>
<td>K = 16, Dinamik Yük Dengeleme Alg, VE</td>
</tr>
<tr>
<td>26</td>
<td>K = 16, Dinamik Yük Dengeleme Alg, VEYA</td>
</tr>
</tbody>
</table>
Abstract
In information retrieval, the queries are responded in a parallel manner to reduce the average response time and to increase the throughput. The performance of the parallel system is further increased by replicating documents or terms across multiple machines. In a parallel query processing architecture, the partitioning of the index can either be term-based or doc-based. In term-based partitioning, a common and accepted approach for replication is to replicate a certain number of the most frequent terms and their inverted lists across all index servers. In this work, we claim that the query logs and thus term access frequencies can better be utilized for a more intelligent replication scheme in term-based partitioned indexes. For this purpose, we propose a hypergraph model that captures the relationship between queries and the terms it requests. Furthermore, we propose two replication approaches to replicate the vertices of a hypergraph: one-phase and two-phase. In one-phase approach, replication is achieved together with the partitioning. We use an FM-based (Fiduccia-Matheyse) two-way replication heuristic to achieve replication for two-way partitionings. Then we use this heuristic in a recursive bipartitioning framework to achieve K-way partitions. In two-phase approach, we perform replication after obtaining a K-way partition without replication. We utilize various concepts such as DM (Dulmage-Mendelsohn) decomposition and ILP (Integer Linear Programming) formulation for replicating vertices. With experiments, we show that both of the methods are effective tools for improving cutsize values of the partitions using a small amount of replication. For the one-phase approach, we developed a tool called rpPAToH by extending PaToH that can be used generically. Finally, we test rpPaToH in a real time parallel query processing system which is implemented again by us, called repl-ABCServer. We show that the overall communication volume of the parallel query processing system can greatly be reduced, and thus, the throughput and the average response time can greatly be improved.

Keywords: Parallel Query Processing, Hypergraph, Replication, Heuristic
Özet


Anahtar Kelimeler: Paralel Sorgu İşleme, Hiperçizge, Çoklama, Sezgisel
1 GİRİŞ

1.1 Paralel Sorgu İşleme Sistemleri
Klasik bir arama motorunun yapısı Şekil 1’de verilmiştir. Bu yapıda, Ağ üzerindeki sayfalar ağ tarayıcıları denen programlar tarafından getirilmekte ve lokal diske kaydedilmektedir. Kaydedilen bu sayfalar daha sonra bir paralel metin işleme sistemi tarafından kullanıcılara sunulmaktadır.

![Şekil 1: Klasik bir arama motoru yapısı.](image)

Basit bir metin işleme sisteminin temel görevi kullanıcı sorgularını işlemek ve kullanıcılara sorguları ile ilgili dökümanları sunmaktadır. Büyük koleksiyonlarda etkin sorgu işleme için ara bir temsil sistemi (örneğin indeksleme sistemi) gerekmektedir. Ters dizin sistemleri bu amaçla kullanılmaktadır.
1.1.1 Ters Dizin Yapısı
Elde bulunan döküman koleksiyonundaki farklı kelime sayısı K ise, bu koleksiyondan oluşturulan ters dizin I = {I_1, I_2, ..., I_K}, K adet ters listeden (inverted list) oluşur. Bir ters dizinin indeks kısmını genellikle küçüktür ve ana belleğe sığar ama ters listeler genel olarak diske tutulurlar. Her bir ters liste tek bir terim ile ilgilidir ve bu terimi içeren dökümanlarla ilgili kayıtları (postings) tutarlar. Bir kayıttı ise kaydın temsil ettiği dökümanın numarası ve kaydın geçtiği ters listenin temsil ettiği terimin dökümandaki ağırlığı tutular. Bu ağırlık döküman ile o terim arasındaki ilişkinin derecesini gösteren bir ölçütür.

1.1.2 Sorgu İşleme


Bir kullanıcı sorgusunu işlemeye çeşitli aşamada gerçekleştirir. Sorgu içerisindeki terimler için öncelikle bu terim ile ilgili tüm kayıtları diskten getirilir, bu listenin içindeki bütün döküman kayıtları incelenir, ilişkili dökümanlar benzerlik skorlarına göre büyükten küçüğe doğru sıralanılır ve en alakalı dökümanlar kullanıcıya gösterilir.

1.1.3 Paralel Sistemlerde Ters Dizin Organizasyonları

1.1.4 Paralel Sorgu İşleme
Bir paralel metin işleme sisteminde bir sorgunun işlemesi çeşitli aşamada gerçekleştirilir ve bu aşamalar uygulanılan paylaştırma yönteminin terim tabanlı ya da döküman tabanlı olmasına göre farklılıklar gösterir. Terim tabanlı paylaştırmada bir sorgu terimi ile ilgili bütün kayıtlar tek bir sorgu
işlemcisine verildiği için merkezi simsar kullanıcı sorgusunun alt sorgulara (subqueries) ayrıştırır ve her bir alt sorguyu o alt sorgudaki terimlerden sorumlu olan sorgu işlemlerine gönderir. Döküman tabanlı paylaşımı yönteminde ise bir sorgu terimi ile ilgili kayıtlar pek çok sorgu işlemeçine dağılmış olabileceği için her bir sorgu işlemeçine kullanıcı sorgusunun tamamı gönderilir.


1.1.5 Paralel Sistemlerde Ters Dizin Çoklaması

1.2 Hiperçizge Bölümleme Problemi
Verilen bir \( \Pi \) bölümlemesinde herhangi bir bölümün ağırlığı o bölümdeki düğümlerin yükleri toplamıdır. Eğer bölümlemenin bölümlerinden birinde bir hiperkenarın bacağı varsa hiperkenar o bölüme bağlıdır denir. Bir \( n_i \) hiperkenarının bağlılık kümesi (connectivity set) \( \Lambda(n_i) \), hiperkenarın bağlı olduğu tüm bölümlerin kümesidir. Bu kümedeki bölüm sayısı \( \lambda(n_i) = |\Lambda(n_i)| \) hiperkenarın bağlılığını (connectivity) verir. Bağlılığı birden fazla olan hiperkenara kesittedir denir.

Tüm bu tanımlamalardan sonra, P-parçalı hiperçizge bölümleme problemi, H hiperçizgesi için, bölümlerin ağırlıkları arasındaki dengeyi sağlayan ve kesitte bulunan hiperkenarlar üzerinde tanımlı bir paha fonksiyonunu en aza indiren bir P-parçalı \( \Pi = \{V_1,..,V_p,..,V_P\} \) bölümlemesi bulmak olarak ifade edilebilir. Sıkça kullanılan fonksiyonlardan \( (\lambda) \) bağlılık ölçütü şu şekilde hesaplanmaktadır:

\[
\text{Paha}(\Pi) = \sum \lambda(n_i) \times \text{Paha}(n_i), \quad \forall n_i \in N.
\]


### 1.3 Katkılar


Proje çıktlarının genelleştirilmiş ekibimiz tarafından geliştirilmiş olan PaToH hiperçizge bölümleme kütüphanesi içerisinde çoklama kullanarak hiperçizge bölümleme yetisi eklenmiş ve
paralel hesaplama konusunda kullanımı olan bir çoklama kütüphanesi (rpPaToH) geliştirilmiştir. Buna ek olarak, rpPaToH kullanılarak elde edilen bölümler gerçek zamanlı bir paralel sorgu işleme sisteminde kullanılarak önerilen algoritmaların geçerliliği deneylerle kanıtlanmıştır.

2 LİTERATÜR ÖZETİ


İyimser çoklama mimarileri günümüz dağıtık (distributed) servis uygulamalarında sıkılkla kullanılmaktadır. Bu alanda en bilindik uygulamalar: DNS, Usenet, PDA ve Bayou iletişim mimarileri (SAITO, 2005). Benzer şekilde kötümsers çoklamaya örnek olacak olursak, bu sınıfa anlık veri tutarlılığının olmadığı bir kriter olduğu saha çalışması (bankacılık, finans, vs.)
girmektedir. Kötümser çaklama en temel örnekler arasında ACID uyumu RDBMS sistemlerindeki (IBM DB2, Oracle, PostgreSQL, vs.) usta-usta (master-to-master) ve/veya usta-yamak (master-to-slave) çaklama uygulamalarını verebiliriz. Ek olarak, iyimser çaklananın günümüzdeki pratik örneklerine bakacak olursak bunlar arasında arama motorları (Google, Yahoo!, vs.) başı çekmektedir. Bunun yanında, bunların servis olarak sunulduğu (Amazon S3) ticari hizmetler de mevcuttur. Görülmüştür ki, çaklama işleri büyük ölçekli sistemlerde beklenenin dışında problemler ortaya çıkaramaktadır. (Örneğin Facebook'da yaşanan Load Balancer Router ihtiyacı, ya da Amazon S3 servisinde kopyalarda gerçekleşen bir hatanın yaklaşık 6 saat içinde iyimser çaklama dolayısı ile dünya üzerinde diğer tüm sunuculara sıçraması (AMAZON, 2008), vs.) Bu gibi problemler gerçekleştirdiğimim mimarinin uygulamaya dökülmesi esnasında oldukça ehemmiyet taşıyor, geliştiriciler tarafından ayrıca dikkat edilmesi gereken noktalar arasındadır.


çözümü için hiperçizge bölümlmeye dayalı modeller geliştirmiş bulunuyoruz (CAMBAZOGLU, 2006a; CAMBAZOGLU, 2006b; CAMBAZOGLU, 2006c).


3 TEK AŞAMALI FM-TABANLI ÇOKLAMALI HİPERÇIZGE BÖLÜMLEME

Tek aşamalı çoklamalı hiperçizge bölümlene probleminin çözümü için çok-seviyeli hiperçizge bölümlene yöntemi genişletip kullandık. Çoklama, çok-seviyeli yöntemin açılma safhasında arıtma sezgiseli olarak kullanılan Fiduccia-Mattheyses (FM) algoritmasını genişleterek elde

3.1 Çoklamalı (Replicated) FM (rFM)

3.1.1 Tanımlar

2-ışını çoklamalı bir bölülemende, \( \Pi = \{A, B\} \), bir düğüm \( A \), \( B \) ya da çoklanmış durumda ise ikisinde birden bulunabilir. Buna göre bir düğüm \( A \), \( B \), ve \( AB \) durumlarından birinde olabilir. Bu durum algoritmalarımızda State alanı ile ifade edilmektedir.

Bir hiperkenarın \( A \) ve \( B \) ’deki düğüm sayısı \( \sigma_A(n_j) \) ve \( \sigma_B(n_j) \) ile gösterilmektedir. Bu hiperkenarin çoklanmış düğüm sayısı ise benzer bir şekilde \( \sigma_{AB}(n_j) \) ile ifade edilmektedir. Örneğimizde bir hiperkenarın başca dağılımı \( \sigma(n_j) = (\sigma_A(n_j); \sigma_B(n_j); \sigma_{AB}(n_j)) \) göstermektedir. Açık bir şekilde, bir hiperkenarin toplam düğüm sayısı, \( |Pins(n_j)| = \sigma_A(n_j) + \sigma_B(n_j) + \sigma_{AB}(n_j) \) toplamına eşittir. Bir hiperkenarın başca dağılımı \( \sigma_A(n_j) > 0 \) veya \( \sigma_B(n_j) > 0 \) koşullarını sağlarsa bu hiperkenara kesit denmektedir. Bir hiperkenarin pahası \( c(n_j) \) ile gösterilmektedir. Bir düğümün hiperkenarları \( Nets(v_i) \) ile ifade edilmektedir.

\[ \text{Şekil 2: } v_1 \text{ düğümünün taşınması ve çoklanması.} \]

Algoritmamız üç değişik işlemi desteklemektedir: taşma, çoklama ve azlama. Taşma ve çoklama işlemleri çoklanmış düğümler için geçerliyken azlama işlemi çoklanmış bir düğümün örnekleri (kopyaları) için geçerlidir. Buna göre her bir düğümün iki tane kazanç değeri olacaktır. Bu kazanç değerleri şunlardır:

- **Taşma kazancı**, \( g_m(v_i) \), bir düğümün tümleyen bölümüne taşınmasıyla elde edilen kazançtır. Basit bir taşıma işlemi örneği Şekil 2’de görülmektedir. \( v_1 \) düğümünün taşınması \( n_1 \) hiperkenarı kesitte getirirken \( n_2 \) hiperkenarı kesitten Kurtarmıştır. Bu durumda \( g_m(v_1) = c(n_2) - c(n_1) \) olmaktadır.

- **Çoklama kazancı**, \( g_r(v_i) \), düğümün tümleyen bölümüne çoklanmasıyla elde edilen kazançtır. Bir düğüm çoklandığından herhangi bir hiperkenarını kesite getirmesi mümkün değildir. Bu, bir düğüm üstündeki taşıma işlemiyle çoklama işlemi arasındaki temel farkı
oluşturmaktaadir. Sonuç olarak, çoklama işleminin kesit pahasını arttırması söz konusu değildir. Şekil 2'de $v_1$ düğümünün A'dan B'ye çoklenmesi görülmektedir. Bu işlem, $n_2$ hiperkenarını kesitten kurtarmaktadır, dolayısıyla, $g_f(v_1) = c(n_2)$ olmaktadır. Üzerinde işlem yapılan düğüm işlemeden sonra kilitlenmekte ve kilitli düğümler örneklerimizde gri renkte gösterilmektedir.

- Azlama kazancı, $g_{u,A}(v_i)$ veya $g_{u,B}(v_i)$, çoklanmış bir $v_i$ düğümünün örneklerinden birinin bulunduğu bölümden silinmesiyle elde edilen kazançtır. Herhangi bir azlama işlemi kesit pahasını azaltamayacağı için, bir kopyanın maksimum azlama kazancı 0'dır. Buna ilaveten, bir azlama işlemi iç bir hiperkenarın kesit pahasını kurtarmak için, çoklanma işlemi sıfır olan kopyaları gerekşiz kopyalar olarak adlandırırmamızı çünkü bunların silinmesi kesit pahasını azaltmayacaktır. Şekil 3'de görüldüğü gibi, bir hiperkenarın kesit pahasını kurtarmak için, çoklanma işlemine gerek yoktur. A bölümden gerekli dış kopya gerekşiz kopya olarak adlandırılmaktadır. Diğer yandan, B bölümden gerekli kopya gerekşiz kopya olup, silinmesi hala hiperkenarının kesit pahasını azaltmamakta. Dolayısıyla, $g_{u,A}(v_1) = -c(n_1)$ ve $g_{u,A}(v_1) = 0$.

Şekil 3: Çoklanmış $v_1$ düğümüne ait iki örneğin azlanması.

3.1.2 Genel rFM Algoritması
3.1.2.1 İşlem Seçimi

Algoritamız çeşitli işlemleri desteklediği için bunlar arasında öncelik tabanlı bir seçim yaklaşımı öneriyoruz. İşlem seçimi, gereksiz kopya sayısını minimize etmek, çoklamayı sınırlamak ve dengeyi geliştirmek gibi ana prensipler üzerine odaklanmaktadır. İşlem önceliği gereksiz kopyalardan hemen kurtulmak için sıfır kazançlı azlama işlemine verilmiştir. Negatif kazançlı azlama işlemleri pahada kötülüğeye yol açacağı için yapılmamaktadır. Eğer sıfır kazançlı azlama işlemi yoksa, taşıma ve çoklama işlemlerinden kazancı yüksek olanı öncelik tanınmaktadır. Eğer taşıma ve çoklama işlemlerinin kazançları eşitse, verilen çoklama miktarını harcamamak için öncelik taşıma işleme verilmektedir. Bunlara ek olarak, sıfır kazançlı çoklamalar gereksiz kopyalar yaratacağı için bu işleme izin verilmemekte, fakat, sıfır kazançlı dengeyi geliştiren taşıma işlemlerine belli bir sayıya kadar izin verilmektedir. Bir \( v_i \) düşümünün çoklama kazancı her zaman taşıma kazancından her zaman eşit ya da daha fazla olacağ için \( g_r(v_i) \geq g_m(v_i) \), genelde, herhangi bir geçişte çoklama işlemleri taşıma işlemlerinin sayısından çok daha fazla olmaktadır. Bu meseleyi *gradyant yöntemle* ele alıyoruz.

3.1.2.2 Gradyant Yötem

Gradyant yöntem çoklamayı destekleyen FM algoritmalarında daha düşük pahalı bölümler elde etmek için öne sürülmüş bir yöntemdir. Bu yöntemdeki ana mantık çoklamaya bir geçişin ilerdeki safhalarında, özellikle taşıma ile ulaşılabilecek pahadaki gelişme belirli bir eşliğin altında düştüğünde izin vermektedir. Bunun sebebi, bir geçişin başında yapılan çoklamanın algoritmanın üzerinde çalıştığı bölümlere değişitirme yeteneğini kısıtlamasıdır. Buna ek olarak, eğer çoklama algoritmanın sonraki safhalarında yapılırsa, algoritma bu şekilde taşıma işlemlerile ulaşılmış yerel minimumdan hareket ederek iyi sonuçlara ulaşabilir. Önerdğımız
algoritmamızda, gradyant yöntemi değiştirip, pahadaki gelişme belli bir eşünün altında düşine kadar taşımak ve azlama işlemlerine öncelik tanıdıktan sonra, çoklama işleme izin veriyoruz.

3.1.2.3 Erken Çıkış
Bu yöntem, FM temelli algoritmaların zaman performansını geliştirmek için sık kullanılan basit bir yöntemdir. Bu yöntemde, eğer önceden belirlenmiş bir sayı kadar işlemde pahada gelişme sağlanamıyorsa, büyük ihtimalle yerel bir minimumda takılmış olan FM algoritmasi sonlandırılır.

3.1.3 Hiperkenar Kritikliği
Algoritmamız VLSI literatüründeki FM algoritmalarıyla kıyaslandığında, onlara nazaran daha fazla yükün güncellemesi içermektedir. Bunun sebebi her düğümün iki kazanç değerine sahip olması ve her kazanç güncellemesinde bu iki kazanç değerinin bir ya da ikisinin güncellemesidir.


Taşıma işlemi için kritik hiperkenar tanıımı aşağıdaki gibidir:

\[ \text{eğer } \sigma_A(n_j) = 1 \text{ veya } \sigma_B(n_j) = 0 \text{ ise } n_j \text{ A bölümüne taşımak-kritik} \]
\[ \text{eğer } \sigma_B(n_j) = 1 \text{ veya } \sigma_A(n_j) = 0 \text{ ise } n_j \text{ B bölümüne taşımak-kritik} \]

Çoklama işlemi için:

\[ \text{eğer } \sigma_A(n_j) = 1 \text{ ise } n_j \text{ A bölümüne çoklama-kritik} \]
\[ \text{eğer } \sigma_B(n_j) = 1 \text{ ise } n_j \text{ B bölümüne çoklama-kritik} \]

Azlama işlemi için:

\[ \text{eğer } \sigma_B(n_j) = 0 \text{ ve } \sigma_A(n_j) > 0 \text{ ise } n_j \text{ A bölümüne azlama-kritik} \]
\[ \text{eğer } \sigma_A(n_j) = 0 \text{ ve } \sigma_B(n_j) > 0 \text{ ise } n_j \text{ B bölümüne azlama-kritik} \]

Görüldüğü üzere, çoklama ve azlama işlemlerindeki kritik hiperkenar tanımı, taşma işlemindeki kritik hiperkenar tanımını sağlayan koşulların bir alt kümesidir. Fakat, azlama işleminde bir
istisna oluşmaktadır. Bu, bir $n_j$ hiperkenarının eğer hiç kopyalanmamış bacağı yoksa ortaya çıkan bir durumdur.

Algorithm 2: İlk taşıma, replikasyon ve unreplikasyon kazanç değerlerinin hesaplanması

```
Input: $H = (V, N), E^R = \{A, B\}$
1 foreach $v_i \in V$ do
2     if $State(v_i) \neq AB$ then
3         $g_m(v_i) \leftarrow c(Nets(v_i))$
4         $g_r(v_i) \leftarrow 0$
5     else
6         $g_{uA}(v_i) \leftarrow 0$
7         $g_{uB}(v_i) \leftarrow 0$
8 foreach $n_j \in N$ do
9     foreach $v_i \in Pins(n_j)$ do
10        if $State(v_i) \neq AB$ and $n_j$ kesitte ise then
11           if $\sigma_A(n_j) = 1$ ve $State(v_i) = A$ veya $\sigma_B(n_j) = 1$ ve $State(v_i) = B$ then $n_j$ A ve B bölümüne kritik
12              $g_m(v_i) \leftarrow g_m(v_i) + 2 \cdot c(n_j)$
13              $g_r(v_i) \leftarrow g_r(v_i) + c(n_j)$
14           else
15              $g_m(v_i) \leftarrow g_m(v_i) + c(n_j)$
16        else if $State(v_i) = AB$ ve $n_j$ içinde ise then
17           if $\sigma_A(n_j) > 0$ ve $\sigma_B(n_j) = 0$ then $n_j$ A bölümüne kritik
18              $g_{uA}(v_i) \leftarrow g_{uA}(v_i) - c(n_j)$
19           else if $\sigma_B(n_j) > 0$ ve $\sigma_A(n_j) = 0$ then $n_j$ B bölümüne kritik
20              $g_{uB}(v_i) \leftarrow g_{uB}(v_i) - c(n_j)$
```

3.1.4 rFM Algoritmasının Detayları

3.1.4.1 İlk Kazanç Hesabı
Algoritma 2’deki ilk kazanç değerlerinin hesaplanması iki ana döngüden oluşmaktadır. İlk döngü düğümler üzerinde gezerek ilk kazanç değerlerini resetlerken (1-7 satırları), ikinci döngü bütün hiperkenarların bacakları üstünden gezerek kazançların ilklendirilmesini tamamlamaktadır.

İlk olarak, çoklanmamış düğümlerin taşıma ve çöklama kazançları mümkün olan en küçük değerlerine atanmıştır. Eğer bir $n_j$ hiperkenar kesitte ise, onun bacaklarının taşıma ve çöklama değerlerinin güncellenmesi gerekebilir. Eğer bu kesitte olan $n_j$ hiperkenar bir bölüm taşıma-veya çöklama-kritikse, $n_j$’nin kritik olduğu bölümdeki bacaklarının taşıma ve çöklama kazançlarının güncellenmesi gerekmektedir. Bu düğümlerin taşıma kazançları $c(n_j)$ kadar artırılırken, çöklama kazançları $2c(n_j)$ kadar artırılmaktadır.

Taşıma ve çöklama kazançlarının aksine, azlama kazançlarına ilk olarak mümkün olan maksimum değerleri verilmiştir. Eğer bir $n_j$ hipekenari azlama-kritikse (ve dolayısıyla içteyse),
bu hiperkenarın bacaklarında bulunması mümkün kopyaların azlama kazançlarının güncellenmesi gerekebilir. İşte bulunan bu hiperkenarla aynı bölümde bulunan kopyaların azlama kazanç değerleri, eğer bu bölümde bu hiperkenara ait en az bir tane çoklanmamış düğüm var ise, $c(n_j)$ kadar azaltılmalıdır.

Şekil 4’te örnek bir hiperçizge ve bu hiperçizgedeki düğümlerin kazanç değerleri ile hiperkenarların bacak dağılımları görülmektedir. Şekilde görüldüğü üzere $n_4$, $n_5$ ve $n_6$ hiperkenarları kesitte olup, eğer bu hiperkenarların pahaları birim paha olarak kabul edilirse bu bölümlemenin toplam pahası 3 olmaktadır.

Şekil 4: Örnek hiperçizgedeki düğümlerin ilk kazanç değerleri ile hiperkenarların bacak dağılımları.

3.1.4.2 Taşıma işleminde sonraki kazanç güncellemesi
Algoritma 3, verilen bir $v^*$ düğümünün $A$ bölümünden $B$ bölümüne taşınmasından sonra hiperkenarların bacak dağılımlarının ve $v^*$'in komşularının kazanç değerlerinin güncellenmesini kapsamaktadır. Algoritma, $v^*$'a ait alanların, $\text{Nets}(v^*)$ kümesindeki hiperkenarların bacak dağılımlarının ve $v^*$'in komşularının kazanç değerlerinin güncellenmesini kapsamaktadır. Her bir $n_j \in \text{Nets}(v^*)$ için, bacak dağılımları $\sigma_A(n_j)$ bir azaltılmaktak ve $\sigma_B(n_j)$ bir artırılmaktadır. Bir hiperkenarin bacak dağılımı değiştiğinde, aynı zamanda kritikliği de değişebilmektedir.

Algoritma 3'te önce $\sigma_A(n_j)$ değerleri hiperkenarların kritiklik koşullarını değişip değişmediğini görmek için kontrol edilmekteidir. Eğer $\sigma_A(n_j) = 0$ olmuşsa, $n_j$ B bölümüne iç olarak bu bölüme taşıma- ve azlama-kritik olmuştur. Bu durumda B'deki düğümlerin taşıma ve azlama kazanç değerleri $c(n_j)$ kadar azaltılmaktadır. Öte yandan, eğer $\sigma_A(n_j) = 1$ ise, $n_j$ A bölümüne taşıma-
ve çoklama-kritik olmuştur ve şimdi A’da bulunan tek düğüm n_j’yi kesitten kurtarabilmektedir. Bu bölümdeki tek düğümün taşıma ve çoklama kazanç değerleri c(n_j) kadar arttırılmalıdır.

Daha sonra σ_B(n_j) değeri bir arttırdıktan sonra, bu güncellemeden dolayı kaynaklanan kritiklik koşullarının sağlanıp sağlanmadığı kontrol edilmektedir. Eğer σ_B(n_j) = 1 olmuşsa, bu n_j'nin A bölümüne önceden iç olduğu anlamına gelmektedir. Şimdi, n_j kesittedir ve bu hiperkenarın A bölümündeki bacakları artık onu kesip getiremeyecektir. Dolayısıyla, A bölümündeki düğümlerin taşıma ve azlama kazanç değerleri c(n_j) kadar arttırılmalıdır. Eğer ise, bu, hiperkenarının B bölümüne taşıma- ve çoklama-kritik olduğu anlamına gelir. Bu durumda B bölümündeki tek düğümün, eğer kilitlememişse, taşıma ve çoklama kazançları c(n_j) kadar azaltılmalıdır.

Şekil 4'te, işlem seçme kriterleri göz önüne alındığında, sıfır kazanç değerli azlama olmadığı için taşıma ya da çoklama işlemlerinden birini seçmek durumundayız. En yüksek kazançlı işlemin değeri 1'dir ve bu durumda taşıma ve çoklama işlemlerinin kazanç değerleri eşittir. Öyleyse işlem seçme kriterlerine göre seçilecek düğüm v_4 ve işlem ise taşıma işlemidir. Şekil 5 v_4 düğümünün taşınmasından sonraki bölümleme için kazanç değerleri ve bacak dağılımlarını
göstermektedir. Bu taşıma işlemi sonucunda $n_5$ hiperkenarı kesitten kurtulmuştur. Sonuçta toplam paha 2’ye inmiştir.

Şekil 5: $v_4$ düğümünün taşınmasından sonra kazanç değerleri ve bacak dağılımları.

```
Algorithm 4: $v^*$’ın $A$ dan $B$ ye çöklenmesinden sonraki kazanç güncellemeleri

Input: $H = (\mathcal{V}, \mathcal{N}), H_2 = \{V_A, V_B\}, v^* \in V_A$

1. $State(v^*) \leftarrow AB$
2. $v^*$’ı kilitle
3. foreach $n_j \in Nets(v^*)$ do
4. \hspace{1em} $\sigma_A(n_j) \leftarrow \sigma_A(n_j) - 1$
5. \hspace{1em} $\sigma_{AB}(n_j) \leftarrow \sigma_{AB}(n_j) + 1$
6. \hspace{1em} if $\sigma_A(n_j) = 0$ then \hspace{0.5em} $n_j \text{ } B$ bölümüne kritik olduğu
7. \hspace{1.5em} foreach $\text{kilitlenmemiş } v_j \in Pins(n_j)$ do
8. \hspace{2em} if $State(v_j) = B$ then
9. \hspace{3em} $g_m(v_j) \leftarrow g_m(v_j) - c(n_j)$
10. \hspace{3em} if $\sigma_B(n_j) = 1$ then
11. \hspace{4em} $g_r(v_j) \leftarrow g_r(v_j) - c(n_j)$
12. \hspace{3em} else if $State(v_j) = AB$ then
13. \hspace{4em} if $\sigma_B(n_j) = 0$ then
14. \hspace{5em} $g_{n,A}(v_j) \leftarrow g_{n,A}(v_j) + c(n_j)$
15. \hspace{4em} else if $\sigma_B(n_j) > 0$ then
16. \hspace{5em} $g_{n,B}(v_j) \leftarrow g_{n,B}(v_j) - c(n_j)$
17. \hspace{3em} else if $\sigma_A(n_j) = 1$ then \hspace{0.5em} $n_j \text{ } A$ bölümüne kritik oldu
18. \hspace{4em} foreach $\text{kilitlenmemiş } v_j \in Pins(n_j)$ do
19. \hspace{5em} if $State(v_j) = A$ then
20. \hspace{6em} $g_m(v_j) \leftarrow g_m(v_j) + c(n_j)$
21. \hspace{6em} if $\sigma_B(n_j) > 0$ then
22. \hspace{7em} $g_r(v_j) \leftarrow g_r(v_j) + c(n_j)$
```
3.1.4.3 Çoklama İşleminde Kazanç Güncellemesi

Algoritma 4, verilen bir $v^*$ düğümünün $A$ bölümünden $B$ bölümüne çoklanmasından sonra gereklidir alan, bacak dağılımlarına ve kazanç güncellemelerine göstermektedir. Algoritma öncelikle çoklanan düğümün durumunun değiştirilmesi ve bu düğümün iki örneğinin de kilitlenmesiyle başlamaktadır. Daha sonra $v^*$'nin her bir hiperkenarının $\sigma_A(n_j)$ değerleri bir azaltılıp $\sigma_{AB}(n_j)$ değerleri de bir artırıldktan sonra gereklidir kazanç güncelleme işlemlerini yapmaktadır.

Çoklanacak $v^*$ düğümünün ait hiperkenarların $\sigma_A(n_j)$ değerleri azaltıldiktan sonra, bu bacak dağılıminin değişiminin yol açtığı kritiklik değişşim koşulları kontrol edilmektedir. Eğer $\sigma_A(n_j) = 0$ olmuşsa, bu $n_j$'nin $B$ bölümüne taşıma- ve azlama-kritik olduğu anlamına gelmektedir. Bu durumda $n_j$'ye bağlı düğümler ve kopyaların taşıma ve azlama kazancı $c(n_j)$ kadar azaltılmalıdır. $\sigma_B(n_j)$ değer değişmediği için bazı istisnai durumlar ortaya çıkmaktadır ve bu durumlar da halledilmektedir. Diğer taraftan $\sigma_A(n_j) = 1$ ise, $n_j$ $A$ bölümüne taşıma- ve çoklama-kritik olmuştur. Bu durumda $n_j$'ye bağlı tek çoklanmamış düğüm simdi onu kesitten kuratarabilir ve dolayısıyla bu düğümün taşıma ve çoklama kazancı $c(n_j)$ kadar artırılmalıdır.

Şekil 6: $v_6$ düğümünün çoklanmasından sonra kazanç değerleri ve bacak dağılımları.

$v_4$ düğümünün taşınmasından sonra şimdi üzerinde işlem yapılacak yeni bir düğüm seçilecektir. Şekil 5'te en yüksek kazanca sahip (1) iki tane düğüm vardır ve bu düğümler $v_5$ ve $v_6$ düğümlerinin çoklanmasıdır. Bu iki düğümden $v_6$'nin çoklanmasını seçiyoruz. Şekil 6 $v_6$ düğümünün çoklanmasından sonra kazanç değerleri ve bacak dağılımları.
düğümünün çoklanmasından sonraki bölümlene için kazanç değerleri ve bacak dağılımlarını göstermektedir. Bu işlem sonucunda \( n_4 \) hiperkenarı kesitten kurtarılmış ve şekildeki bölümlenenin toplam pahası 1 olmuştur.

\[
\text{Algorithm 5: } v^* \text{'in } A \text{ bölümden unreplike edildikten sonra } v^* \text{'in kazanç güncellemeleri}
\]

\[
\begin{align*}
\text{Input: } & (V, N), H_2 = \{V_A, V_B\}, v^* \in V_A \\
1 & \text{State}(v^*) \leftarrow B \\
2 & v^*' \text{ kilitle} \\
3 & \text{foreach } n_j \in Nets(v^*) \text{ do} \\
4 & \quad \sigma_B(n_j) \leftarrow \sigma_B(n_j) + 1 \\
5 & \quad \sigma_{AB}(n_j) \leftarrow \sigma_{AB}(n_j) - 1 \\
6 & \quad \text{if } \sigma_B(n_j) = 1 \text{ then } \triangleright n_j A \text{ bölümine kritiktir} \\
7 & \quad \text{foreach } \text{ kilitlememiş } v_i \in Pins(n_j) \text{ do} \\
8 & \quad \text{if } \text{State}(v_i) = A \text{ then} \\
9 & \quad \quad g_m(v_i) \leftarrow g_m(v_i) + c(n_j) \\
10 & \quad \quad \text{if } \sigma_A(n_j) = 1 \text{ then} \\
11 & \quad \quad g_r(v_i) \leftarrow g_r(v_i) + c(n_j) \\
12 & \quad \quad \text{else if } \text{State}(v_i) = AB \text{ then} \\
13 & \quad \quad \quad \text{if } \sigma_A(n_j) = 0 \text{ then} \\
14 & \quad \quad \quad \quad g_{n,B}(v_i) \leftarrow g_{n,B}(v_i) - c(n_j) \\
15 & \quad \quad \quad \quad \text{else if } \sigma_A(n_j) > 0 \text{ then} \\
16 & \quad \quad \quad \quad \quad g_{n,A}(v_i) \leftarrow g_{n,A}(v_i) + c(n_j) \\
17 & \quad \quad \quad \quad \text{else if } \sigma_B(n_j) = 2 \text{ then } \triangleright n_j B \text{ bölümine kritiktir} \\
18 & \quad \quad \quad \quad \text{foreach } \text{ kilitlememiş } v_i \in Pins(n_j) \text{ do} \\
19 & \quad \quad \quad \quad \text{if } \text{State}(v_i) = B \text{ then} \\
20 & \quad \quad \quad \quad \quad g_m(v_i) \leftarrow g_m(v_i) - c(n_j) \\
21 & \quad \quad \quad \quad \quad \text{if } \sigma_A(n_j) > 0 \text{ then} \\
22 & \quad \quad \quad \quad \quad \quad g_r(v_i) \leftarrow g_r(v_i) - c(n_j)
\end{align*}
\]

3.1.4.4 Azlama İşleminden Sonra Kazanç Güncellemesi

Algoritma 5, verilen bir \( v^* \)'ın kopyasının \( A \) bölümden azlandktan sonraki kazanç güncellemelerini göstermektedir. Öncelikle \( v^* \)'ın durumu \( B \) olarak değiştirilip kilitlenmektedir. Daha sonra her bir \( n_j \in Nets(v^*) \) için, \( \sigma_B(n_j) \) değeri bir arttırılmaktak ve \( \sigma_{AB}(n_j) \) değeri bir azaltılmaktadır. Sonra, kritiklik koşulları değişen hiperkenarlar kontrol edilmekte ve gerekli kazanç güncellemeleri yapılmaktadır.

Aznalanacak \( v^* \)'a ait hiperkenarların \( \sigma_B(n_j) \) değerleri bir arttırılduktan sonra bu güncellemeden kaynaklanan kritiklik koşullarının değişimi için kontrolör yapılmaktadır. Eğer \( \sigma_B(n_j) = 1 \) olmuşsa, bu \( n_j \)'nin önceden \( A \) bölümine taşıma- ve azlama-kritik olduğu anlamına gelmektedir. Bu durumda, \( n_j \) artı bir iç hiperkenar olmadiği için, \( A \) bölümdeki çoklanmış düğümlerin taşıma kazançları ile kopyaların azlama kazançları \( c(n_j) \) kadar arttırılmaktadır. Çoklama işleminde olduğu gibi burada da bazı istisnai durumlar ortaya çıkmakta ve bu durumlar da burada ele alınmaktadır. Diğer taraftan, eğer \( \sigma_B(n_j) = 2 \) ise, bu önceden \( n_j \) hiperkenarının \( B \) bölümine taşıma- ve çoklama-kritik olduğu anlamına gelmektedir. Bu durumda \( n_j \)'nin \( B \) bölümdede iki bacağı bulunmaktadır ve bunlardan birisi kilitli durumda olan \( v^* \)'dir. Diğer
bacağının taşıma ve çoklama kazançları $c(n_j)$ kadar azaltılmalıdır çünkü bu düğüm artık $n_j$'yi kesitten kurtaramayacaktır.

Şekil 7: $v_1$'e ait $B$ bölümündeki örnek azlanmasından sonra kazanç değerleri ve bacak dağılımları.

Şekil 6'ya bakıldığında, sıfır kazancı olan bir azlama işlemi görülmektedir. Bu işlem $v_1$ düğümünün $B$ bölümündeki örneği için olup gereksiz olan bir kopyadan kurtulma işlemidir. Bu kopya silindikten sonra toplam kesit büyüklüğünde bir değişme olmayacağı için toplam kesit yine 1 olarak kalmaktadır. Şekil 7 $v_4$ düğümünün $B$ bölümündeki örneğinin azlanmasından sonraki bölümlemeye ait kazanç değerlerini ve bacak dağılımlarını göstermektedir.

3.2 Çok Seviyeli Yöntem ve Özyinelemeli Bölümleme

Önerdiğimiz algoritmayı başarılı hiperçizge bölümleme aracı PaToH'a entegre etmiş bulunmaktadır. Bu işlem sırasında çıkan olası problemlerden biri açıklama safhasında seviyeler arasında projeksiyon işlemi sırasında ortaya çıkan çok sayıda gereksiz kopyadan kurtulma işlemidir. Bunun sebebi açıklama safhasında bir seviyeden diğerine geçen serbestlik derecelerindeki artıştır. Gereksiz kopyaların silinmesi algoritmamızın uygulanmasında önemli bir şekilde verilen çoklama miktarını harcama ve açılma safhasının sonunda gelecek bölümlemeler için yeni hiperçizgeler oluşturulurken bu yeni hiperçizgelerin gereksiz bir şekilde büyümesine yol açmaktadır. Önerdiğimiz işlem seçme kriterleri bu sorunu çözmektedir çünkü işlem önceliği her zaman gereksiz kopyaların silinmesidir. Böylece her projeksiyondan

\[ g_m(v_2) = -3, \quad g_r(v_2) = 0, \quad g_m(v_3) = -3, \quad g_r(v_3) = 0, \quad g_m(v_5) = 0, \quad g_r(v_5) = 1, \quad g_m(v_7) = -1, \quad g_r(v_7) = 0 \]

\[ \sigma(n_1) = (3 : 0 : 0), \quad \sigma(n_2) = (1 : 0 : 1), \quad \sigma(n_3) = (3 : 0 : 0), \quad \sigma(n_4) = (2 : 0 : 1), \quad \sigma(n_5) = (0 : 2 : 1), \quad \sigma(n_6) = (1 : 2 : 0), \quad \sigma(n_7) = (0 : 1 : 1) \]

\( paha = 1 \)
sonraki arıtma işlemi başında gerekli kopyalar işlem seçme kriterlerine göre hemen silinmektedir.

### 3.3 Kopya veya Bacak Seçimi ve Küme Kapama Sezgisi

Bölümlenme sırasında çöken bir düğüm (terim), bu düğümü bağlayan her bir hiperkenarın (sorgunun) bu çökenmiş düğümlerden hangisinin kullanılacağı problemi de beraberinde getirmektedir. Eğer bir sorguya ait terimlerin ters listeleri çökenmiş durumdaysa, bu çökenmiş terimlerden hangisinin bu sorgu tarafından kullanılacağı belirlenmesi gerektmektedir. Bu genel anlamda iki farklı durum için gerekli olabilir: (i) bölümleme sonrasında hesaplanması gereken kesit pahası (1. aşama) veya (ii) modellenen gerçek-dünya probleminin hiperkenarlar (yani bizim modelimizde sorgular) üstünde çökenmiş terimlerin hangilerini kullanılamayacağını belirleyemiş zorunlu kılmasını (2. aşama). Bölümleme sonrası bacak seçimi, paralel sorgu işlemede kullanılan terim seçimine karşılık gelmektedir ve eğer dikkatli yapılmazsa 1. aşamada fazla paha hesaplanmasına ve ikinci aşamada da sonuç olarak fazla iletişim hacmine sebep olabilir.

Şekil 8: Bölümleme sonrası $n_j$ hiperkenarı için bacak seçimi. (a) Bacak seçiminden önceki ilk bölümleme. (b) Birinci seçim alternatifi, $n_j$nin bağlı olduğu bölüm sayısı 3. (c) İkinci seçim alternatifi, $n_j$nin bağlı olduğu bölüm sayısı 1.

Bölümleme sonrası kesit pahası hesaplanırken çöken düğümlerin hiperkenarlar için hangisinin kullanılacağı seçmesi ve buna paralel olarak paralel sorgu işleme sırasında çökenmiş terimlerin sorgular için hangilerinin kullanılacağı belirlenmesi kesit hesaplanmasında ve paralel sistemin performansında çok büyük etkiler sahip olabilmektedir. Şekil 8 bu seçimde çökenleme olduğu zaman ne kadar önemli olabileceğini göstermektedir. Şekil 8a'da bölüm sayısı üç olan bir bölümledede $v_r, v_s$ ve $v_n$ düğümlerini bağlayan bir $n_j$ hiperkenarı görülmektedir. Bu düğümlerden $v_r$ ve $v_s$ çökenmiş duruma olup her birinin üç örneği bulunurken, $v_n$ ise çökenmamış durumdadır. Bu örnekte bir hiperkenarın bağıldığı bir düğüm için bir seçim yapıldığında, seçilen bu bacak kalın çizgiyle gösterilmiştir. Hiperkenar $n_j$ için bir
seçim alternatifi Şekil 8b’de görülmekte olup, bu alternatifte \(v_r\) ve \(v_s\) sırasıyla \(V_1\) ve \(V_3\) bölümlerinden seçilmiştir. Burada dikkat edilmesi gereken diğer bir nokta ise çoklanmamış düğümler için bir seçim alternatifi olmadığı ve bu gibi düğümlerin seçilme özelliği sadece bir bölüm olduğudur, \(v_n\) düğümü için olduğu gibi. Daha dikkatli yapılan bir düğüm seçimi Şekil 8c’de görülmekte olup burada bütün düğümler \(V_2\) bölümindenden seçilmiştir. Bu seçim sonucu paha değeri bir olarak görülmektedir.

Bu probleme önerdiğimiz algoritma hem bölümleme sonrası çoklanmış düğüm seçimi için kullanılabilmektedir olup hem de paralel sorgu işleme sırasında çoklanmış terimlerin seçimi için de kullanılabilmekektir.

Bu algoritma iki aşamadan oluşmaktadır. \(nr(n_j, k)\) ve \(r(n_j, k)\) değerleri sırasıyla \(n_j\) hiperkenarın \(V_k\) bölümünde bağladığı çoklanmamış ve çoklanmış düğüm sayılarını ifade etsin. Şimdi \(t\) örneklili \(n_j\) tarafindan bağlanan çoklanmış bir \(v_i\) düğümü düşünülüm. Paralel sorgu işleme esnasında ya da son kesit değerinin hesaplanmasında bu çoklanmış düğümün örneklerrinden birini \(n_j\) hiperkenarı için seçmek durumundadır.

Algoritmanın ilk aşaması şu gözlem üzerine dayanmaktadır. Kesitte kalmış ve \(V_k\) bölümünde en az bir çoklanmamış düğüm bağlayan bir \(n_j\) hiperkenarı düşünülüm (\(nr(n_j, k) > 0\)). Eğer \(r(n_j, k) > 0\) ise, bu durumda \(n_j\)’nin \(V_k\) daki çoklanmış bacaklara olan bağlantıları kesit değerini kütüleştirmeden seçilmelektedir çünkü buradaki çoklanmamış düğüm kesiti zaten katkı yapmaktadır. Bu gözlemden yararlanarak, ilk olarak \(n_j\)’ye ait düğümlerin bacakları üstünde gezmekte ve şu ana kadar eğer \(nr(n_j, k) > 0\) ise \(V_k\)’daki seçilmemiş çoklanmış düğümlere ait bacaklar \(n_j\) için seçilmelektedir. Bu işlem sırasında eğer bir \((n_j, v^k_l)\) bacağı çoklanmış durumda bulunan \(v_i\) için seçilmele ise, \(l \neq k\) durumundaki diğer bütün \((n_j, v^k_l)\) bacakları seçimden çıkartılmaktadır. Burada \(v^k_l\) değeri çoklanmış durumda bulunan \(v_i\) düğümünün \(V_k\) bölümündeki örneğini göstermektedir. \((n_j, v^k_l)\) bacağının \(n_j\) hiperkenarı için seçimlerinden sonra \(v_i\) düğümünün örneklerrinin bulunduğu her bir \(V_k\) bölüm için \(r(n_j, k)\) değeri bir azaltılmaktadır.

Birinci aşamanın sonunda, her bir \(n_j\) hiperkenarı için, bu hiperkenarların bağladığı çoklanmamış durumda bulunan her bir düğüm ait bacaklar otomatik olarak seçilmelektedir çünkü bu bacaklar bu çoklanmamış durumda düğümlerin seçilmesi için tek adaydır. Hiperkenar \(n_j\) için \(v_i \in V_k\) için \((n_j, v_i)\) bacağının seçiminde sonra, \(nr(n_j, k)\) değeri bir azaltılmaktadır. Şekil 9’derinde örnek bacak seçimi sezgisinin adımlarını gös- termektedir. Şekil 9’a da iki çoklanmamış \((v_a\) ve \(v_b\) ve dört çoklanmış \((v_r, v_s, v_t\) ve \(v_u\) düğüm bulunmaktadır. Şekil 9b seçim sezgisinin ilk aşamasından sonraki bacak seçimlerini göstermektedir. Seçilmiş bacaklar \(((n_j, v_b); (n_k, v_b); (n_k, v^3_u); (n_k, v^2_r); (n_k, v_u))\) kalan çizgilerle gösterilmister. Şekil 9b’de seçilmeyen bacaklar \(((n_k, v^4_u), (n_k, v^2_r))\) kaldırılmıştır.

Algoritmanın ikinci aşamasının başında tüm \(nr(n_j, k)\) değerleri sıfıra eşittir çünkü çoklanmamış tüm düğümler birinci aşamada seçilmiştir. Eğer bir \(n_j\) hiperkenarı için en az bir \(V_k\) bölümü için \(r(n_j, k) > 0\) ise \(n_j\) için bacak seçim problemi küme örtme problemine indirgenmektedir. Bu durum Şekil 9b’de \(n_j\)’nin bacak seçim işleminde görülmektedir. Bu örnekte, kümeler
S_1 = \{v_r, v_s, v_t\}, S_2 = \{v_r\} ve S_3 = \{v_s, v_t\} olup örtmeye çalıştığımız ana küme ise S = \{v_r, v_s, v_t\}dir. Küme örtme problemi NP-hard olduğu için ikinci aşamada yakınlık değeri ln n + 1 olan basit bir sezgisel kullanıyoruz. Ana hatlarıyla bu sezgisel her bir adımda şu ana kadar örtülmemiş elemanlardan en fazlasını ortan kümeyi seçip sonra örttügü elemanları ana kümeden çıkarmaktadır. Bu algoritma Şekil 9b'de n_j için koşulduğunda Şekil 9c'de görüldüğü üzere S_1 örtén alt küme olarak seçilecektir. Bu seçme işlemine göre (n_j, v_r^1), (n_j, v_s^1), (n_j, v_t^1) bacakları seçilirken (n_j, v_r^2), (n_j, v_s^4), (n_j, v_t^4) bacakları seçimden çıkarılacaktır. Şekil 9c bu seçim sonucunda bölümleri ve bacakları göstermektedir.

Şekil 9: (a) Seçimden önceki ilk bölümelme. (b) Bacak seçim algoritmasının ilk aşamasında yapılan seçimler. (c) Bacak seçim algoritmasının ikinci aşamasında yapılan seçimler.

3.4 Çoklama Miktarı Dağıtımı
Algoritmamız bölümelme sırasında çoklamaya ulaştığı için, bölümelemeler başına verilecek çoklama miktarının bölümeleme sonucunda elde edilecek paha üzerinde bir etkisi olabilir. Bunun için iki çoklama miktarı dağıtımını tekrarı öneriyoruz:

- **Kademe başına dağıtım:** Bu çoklama dağıtım şemasında özyinelemeli bölümeleme metodunun her bir seviyesine eşit miktarda çoklama miktarı verilmektedir. Verilen bu miktar daha sonra belirli bir seviyedeki bölümelemeler arasında eşit olarak dağıtılmaktadır.

- **Bölümeleme başına dağıtım:** Bu şemada ise her bir bölümeleme eşit miktarda çoklama miktarı almakta.
3.5 Bölümlemeye Ait DeneySEL SONUÇLAR

3.5.1 DeneySEL Düzenek

Önerilen çoklama şeması çok seviyeli hiperçizge bölümleme aracı olan PaToH’a entegre edilmiştir. PaToH’un bu modifiye edilmiş versiyonunu rpPaToH olarak adlandırıyoruz. Deneylerde, kabalaştırma (coarsening) ve başlangıç bölümleme (initial partitioning) safhalarında PaToH ve rpPaToH için aynı parametreleri kullandık. Kabalaştırma safhası için aglomeratif kümeleme ve ilk bölümleme safhası için de açgözlü (greedy) hiperçizge büyütme algoritmaları kullanıldı. Dengesizlik sabiti hem PaToH hem de rpPaToH için 0.10 olarak ayarlandık. Artırmada fazında, PaToH için Sınır FM (BFM) ve rpPaToH için önerilen rFM sezgiselleri kullanıldı. Her bir artıma fazında sezgiseller üç geçiş yapmaktadır. Deney sonuçları olarak beş değişik $K$ (16, 32, 64, 128, 256) ve beş değişik $\rho$ (0.05, 0.10, 0.15, 0.20, 0.25) değeri için tablolar ve figürleri veriyorum. Hatırlanacağı üzere, $K$ bölüm sayısı, $\rho$ ise izin verilen çoğaltma miktarı oranını göstermektedir. Bazı tablolarında yer sıkıntısından dolayı $K=16$ için sonuçlar verilmemiştir.

Bütün algoritmalar C dilinde yazılıp gcc’nin O3 opsiyonuyla derlenmiştir. PaToH’un randomize doğası göz önüne alınarak, rapor edilen sonuçlar algoritmaların ve araçların on defa koşulup ortalamalarının alınmasıyla oluşturulmuştur. Deneylerde 32 GB belleğe sahip altı çekirdekli ve 2.1 GHz saat frekanslı AMD Opteron işlemci kullanılmıştır.

<table>
<thead>
<tr>
<th>Veri kümesi</th>
<th>Düğüm</th>
<th>Hiperkenar</th>
<th>Bacak</th>
<th>Ortalama Hiperkenar Büyüklüğü</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>43,244</td>
<td>200,000</td>
<td>800,034</td>
<td>4.0</td>
</tr>
<tr>
<td>VG</td>
<td>146,742</td>
<td>500,000</td>
<td>1,999,540</td>
<td>4.0</td>
</tr>
<tr>
<td>WP</td>
<td>442,063</td>
<td>1,000,000</td>
<td>3,997,741</td>
<td>4.0</td>
</tr>
<tr>
<td>FB</td>
<td>2,880,004</td>
<td>4,000,000</td>
<td>16,001,754</td>
<td>4.0</td>
</tr>
<tr>
<td>AOL</td>
<td>1,339,596</td>
<td>11,867,848</td>
<td>38,907,550</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Tablo 1: Veri kümeleri özellikleri.

her veri kümesinden iki ve altı arasında terim içeren sorgular üretti. Gerçek dünyada sorguların taklit etmek için toplanan veri kümelerinden dökümanlar rastgele seçilmiş, ve bu seçilen dökümlardan sorgu oluşturmak için seçilen terimlerin frekanslarının Zipfian dağılımında olması sağlanmıştır. Bu sorgu kümeleri kullanarak, düğümler terim, hiperkenarlar ise sorgu olacağı biçimde hiperçizgeler üretmiştir. Bu uygulama alanında düğüm çoğlaması kopya seçiminde (bacak seçim) bahsedildiği üzere lüzumsuz disk erişimine sebep olmadığı için, denge kısıtlaması hiperçizgelerdeki her bir terime birim ağırlık atanarak her bir işlemciye atanak terim sayısı dengelemek olarak yorumlanmıştır. Tablo 1’de bu veri kümelerinden üretilen hiperçizgelerin özellikleri görülmektedir. Tek terimli sorgular bu veri kümelerine dahil edilmemişti çünkü bu tür sorgular tek bacaklı hiperkenarların üretilmesine sebep olmaz ve bu hiperkenarlar daha pahaya katkısı etmemektedir.

<table>
<thead>
<tr>
<th>$\rho$</th>
<th>Şema</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>bis+n</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>bis+gn</td>
<td>0.88</td>
<td>0.91</td>
<td>0.89</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>lev+n</td>
<td>0.90</td>
<td>0.90</td>
<td>0.91</td>
<td>0.89</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td>lev+gra</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.83</td>
<td>0.84</td>
</tr>
<tr>
<td>0.10</td>
<td>bis+n</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>bis+gn</td>
<td>0.90</td>
<td>0.92</td>
<td>0.92</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>lev+n</td>
<td>0.91</td>
<td>0.90</td>
<td>0.88</td>
<td>0.87</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>lev+gra</td>
<td>0.84</td>
<td>0.83</td>
<td>0.81</td>
<td>0.82</td>
<td>0.84</td>
</tr>
<tr>
<td>0.15</td>
<td>bis+n</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>bis+gn</td>
<td>0.93</td>
<td>0.93</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>lev+n</td>
<td>0.94</td>
<td>0.89</td>
<td>0.89</td>
<td>0.85</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>lev+gra</td>
<td>0.87</td>
<td>0.83</td>
<td>0.83</td>
<td>0.82</td>
<td>0.82</td>
</tr>
<tr>
<td>0.20</td>
<td>bis+n</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>bis+gn</td>
<td>0.95</td>
<td>0.94</td>
<td>0.95</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>lev+n</td>
<td>0.89</td>
<td>0.87</td>
<td>0.86</td>
<td>0.84</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>lev+gra</td>
<td>0.84</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.80</td>
</tr>
<tr>
<td>0.25</td>
<td>bis+n</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>bis+gn</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>lev+n</td>
<td>0.89</td>
<td>0.84</td>
<td>0.84</td>
<td>0.83</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>lev+gra</td>
<td>0.84</td>
<td>0.81</td>
<td>0.80</td>
<td>0.81</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Tablo 2: bis+n şemasına göre normalize edilmiş dört farklı rpPaToH şemasının bütün veri kümelerine ait paha değerleri ortalamaları.

3.5.2 Performans Değerledirmeleri
Bu bölümde, rpPaToH’la ilgili balans, çocukla kullanımı ve paha gelişimi gibi değerleri ölçüt alan geniş analizlerin sonuçlarını veriyoruz.

Tablo 2 rpPaToH’ta kullanılan dört farklı şemanın bütün veri kümleri için ortalama paha değerlerinin bis+n şemasına göre normalize edilmiş halini göstermektedir. Tablo’dada, lev kademe başına eşit miktarla çocukla yapılan şemayı gösterirken, bis ise her bölümlme başına eşit miktra çocukla yapılan şemayı ifade etmektedir. Gradyan yöntemi kullanan rpPaToH gra ile ifade edilmiş olup, nor rpPaToH’un gradyan yöntemi kullanmadığı şemayı
göstermektedir. Örneğin tablodaki bis+gra ifadesi rpPaToH'un bölümüne başına eşit miktarda çoklama düşen şemayı ve gradyan yöntemi kullandığını göstermektedir. Kademe başına dağılım bölümüne başına dağılımla kıyaslandığında (kiyaslama için belli bir ρ değeri için bis+nor ve lev-nor, veya bis+gra ve lev+gra sıralarını bakınız), görüldüğü üzere kademe başına çoklama miktarını dağıtan şemayı ve gradyan yöntemini kullanmanın ise bölümler arası farklı hesaplamasını sağlar. Dolayısıyla, özüneleme eşiğinin daha derinlerindeki daha küçük hiperçizgelere ait bölümlerlere daha fazla çoklama şansı tanımlanmıştır. Tablodan görüldüğü üzere, gradyan yöntemi kullanlan rpPaToH, gradyan yöntemi kullanmayan rpPaToH'tan daha iyi sonuçlar elde etmektedir (kiyaslama için belli bir ρ değeri için bis+nor ve bis+gra, veya lev+nor ve lev+gra sıralarını bakınız). Bu sonuçları göz önünde aldığımızda, bundan sonraki tablolarda rpPaToH sonuçları verilen lev+gra şemalar kullanılarak elde edilen sonuçlar, diğer bir deyişle çoklama miktarının kademe başına dağıtıldığı ve gradyan yönteminin kullanıldığı rpPaToH'un sonuçları verilmektedir.


<table>
<thead>
<tr>
<th>Ver Kumesi</th>
<th>K = 32</th>
<th>K = 64</th>
<th>K = 128</th>
<th>K = 256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ρ</td>
<td>%LI</td>
<td>%RU</td>
<td></td>
</tr>
<tr>
<td>bis+nor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>5.97</td>
<td>9.73</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>5.17</td>
<td>9.52</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>4.38</td>
<td>9.31</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>4.06</td>
<td>9.21</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>3.56</td>
<td>9.10</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>lev+nor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>6.82</td>
<td>8.79</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>6.02</td>
<td>8.52</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>5.23</td>
<td>7.85</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>4.78</td>
<td>7.03</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>4.05</td>
<td>7.44</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>bis+gra</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>6.81</td>
<td>8.90</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>6.14</td>
<td>8.44</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>5.44</td>
<td>9.40</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>4.72</td>
<td>9.19</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>4.30</td>
<td>7.23</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>lev+gra</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>9.00</td>
<td>5.74</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>8.58</td>
<td>5.60</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>7.90</td>
<td>5.16</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>7.46</td>
<td>5.00</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>6.84</td>
<td>4.00</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>A2L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>5.54</td>
<td>4.19</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>4.75</td>
<td>7.75</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>4.15</td>
<td>6.55</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>3.80</td>
<td>6.10</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td>3.69</td>
<td>4.44</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>

Tablo 3: PaToH ve rpPaToH için yük dengesizliği (%LI) ve çoklama kullanım (%RU) yüzde.

31
Tablo 3 PaToH+MF ve rpPaToH’a ait yük dengesizliği yüzdelерini (%LI) ve çoklama kullanım yüzdelerini göstermektedir (%RU). Bu değerler aşağıdaki gibi hesaplanmaktadır:

\[ \%LI = \frac{W_{\text{max}} - W_{\text{avg}}}{W_{\text{max}}} \times 100, \]

\[ \%RU = \frac{\sum_{k=1}^{K} W(V_k) - W(V)}{\rho W(V)} \times 100. \]

Bu eşitliklerde, \( W_{\text{max}} \) en ağır yüklü bölümü, \( W_{\text{avg}} \) ortalama bölümü yükünü, \( W(V_k) \) \( V_k \) bölümü için yükünü ve \( W(V) \) ise çoklama dahil edilmiş bir hiperçizgedeki toplam düşüm ağırlığını göstermektedir. Tablo 3’teki dengeler, iki algoritma da izin verilen değerler içinde kalan denge değerleri üretimde ve verilen çoklama oranını artırmak için, elde edilen bölümlerin denge değerleri de iyileşmektedir. Bunun sebebi çoklamanın elde edilen bölümlerin dengesini iyileştirmede bir yan araç olarak kullanılmaktadır.

Tablo 3’teki PaToH+MF ve rpPaToH’da elde edilen çoklama miktarı, %100 kullanmaktadır. Tablo 3’teki algoritmaların karşılaştırıldığında, belli bir kazanan yoktur. Bunun dışında, verilen çoklama oranı arttıkça, elde edilen bölümün denge değerleri de iyileşmektedir. Bunun sebebi çoklamanın elde edilen bölümlerin dengesini iyileştirmede bir yan araç olarak kullanılmaktadır.

Tablo 4: PaToH+MF ve rpPaToH için paha değerleri ($x 10^3$).

Tablo 5 rpPaToH'un PaToH üzerine tüm veri kümeleri için normalize edilmiş ortalama paha değerleri.

Şekil 10a ve 10b PaToH+MF ve rpPaToH'un artan $K$ ve $\rho$ değerleri için göreceli paha değerlerini karşılaştırmaması daha iyi bir şekilde göstermek için sunulmuştur. Şekil 10a da görülüşü üzere, artan $K$ değerleri için, PaToH+MF ve rpPaToH arasında performans farkı CG, VG, ve FB veri kümeleri için azalırken WP ve AOL veri kümeleri için artmaktadır. Şekil 10b'de görülüşü üzere, artan $\rho$ değerleri için, PaToH+MF ve rpPaToH arasında performans farkı FB için artarken, WP ve AOL için azalmakta, ve CG ve VG veri kümeleri için hemen hemen sabit kalmaktadır.

Tablo 6 rpPaToH'un bütün veri kümeleri için PaToH'a göre normalize edilmiş ortalama koşma zamanlarını göstermektedir. Sabit bir $\rho$ değeri için, $K$ arttıkça rpPaToH'un PaToH'a göre verdiği artış miktarı azaldığı için bu deneysel bulgu $K$ arttıkça paha değerinin artmasına ve dolayısıyla paha değerindeki sabit bir çoğalma miktarına bağlı orantısal düşüşün azalmasına bağlıdır.

33
bunun sebebinin başlıca kabalaştırma ve ilk bölütleme fazları esnasında meydana gelen zaman farklarından ileri geldiğini gösteriyor. rpPaToH’ta her bölütlemeden sonra, ileriki seviyelerdeki kabalaştırma ve ilk bölütleme fazları genel olarak PaToH’a göre daha büyük hiperçizgeler üzerinde çalışmak durumunda kalmaktadır. $K$ değeri arttıkça, bu daha büyük hiperçizgeler daha fazla bölülenmek durumunda kalmaktadır ve dolaysıyla rpPaToH ve PaToH arasında fark $K$ arttıkça büyümektedir. Buna rağmen, en büyük $\rho=0.25$ ve $K=256$ değerleri için bile, rpPaToH PaToH’tan ortalama sadece 2.78 kat daha yavaşştır.

Şekil 10a-b: Artan $K$ (10a) ve artan $\rho$ (10b) değerleri için PaToH+MF ve rpPaToH için paha değerleri.
Tablo 5: rpPaToH’un PaToH’a göre bütün veri kümelerindeki ortalama yüzde paha azalma değerleri.

<table>
<thead>
<tr>
<th>ρ</th>
<th>K = 16</th>
<th>K = 32</th>
<th>K = 64</th>
<th>K = 128</th>
<th>K = 256</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>76.76</td>
<td>68.28</td>
<td>60.69</td>
<td>52.47</td>
<td>44.64</td>
</tr>
<tr>
<td>0.10</td>
<td>81.01</td>
<td>72.75</td>
<td>65.02</td>
<td>56.87</td>
<td>49.36</td>
</tr>
<tr>
<td>0.15</td>
<td>83.98</td>
<td>75.80</td>
<td>68.30</td>
<td>60.32</td>
<td>53.11</td>
</tr>
<tr>
<td>0.20</td>
<td>86.19</td>
<td>78.43</td>
<td>71.02</td>
<td>63.06</td>
<td>55.95</td>
</tr>
<tr>
<td>0.25</td>
<td>87.69</td>
<td>80.55</td>
<td>73.42</td>
<td>65.42</td>
<td>58.37</td>
</tr>
</tbody>
</table>

Tablo 6: rpPaToH’un PaToH’a göre normalize edilmiş bütün veri kümelerindeki ortalama koşma zamanı değerleri.

<table>
<thead>
<tr>
<th>ρ</th>
<th>K = 16</th>
<th>K = 32</th>
<th>K = 64</th>
<th>K = 128</th>
<th>K = 256</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>1.76</td>
<td>1.90</td>
<td>2.03</td>
<td>2.20</td>
<td>2.40</td>
</tr>
<tr>
<td>0.10</td>
<td>1.78</td>
<td>1.94</td>
<td>2.17</td>
<td>2.37</td>
<td>2.55</td>
</tr>
<tr>
<td>0.15</td>
<td>1.83</td>
<td>1.97</td>
<td>2.28</td>
<td>2.41</td>
<td>2.58</td>
</tr>
<tr>
<td>0.20</td>
<td>1.82</td>
<td>1.99</td>
<td>2.36</td>
<td>2.55</td>
<td>2.70</td>
</tr>
<tr>
<td>0.25</td>
<td>1.91</td>
<td>2.03</td>
<td>2.42</td>
<td>2.55</td>
<td>2.78</td>
</tr>
</tbody>
</table>

4 İKİ AŞAMLI DM-TABANLI ÇOKLAMALI HİPERÇIZGE BÖLÜMLEME

4.1 İki Parçalı DM-Tabanlı Çözümleme
Önerilen iki aşamalı yaklaşımın birinci aşamasında, K dizin sunuculu bir sistemde, 106E069 numaralı TÜBİTAK projesi kapsamında hem term tabanlı hem de döküm tabanlı ters dizin bölümlemesi problemlerinin çözümü için geliştirmiş olduğumuz hiperçizge modelleri aynen kullanılarak (çoklama hiç düşünülmümeden) ters dizin 2 parça bölümlenir. İkinci aşamada ise her iki bölüm arasındaki hiperkenar kesiti, çičlama işlemini iki bölüm sınırları ile sınırlayarak en aza indirgenmeye çalışılır. Daha sonra bu işlem yinelemeli olarak tekrarlanarak K parçalı çoklamalı bölümlemeler elde edilir. İkinci aşamada hiperkenar kesiti ile bölüm sınırları arasındaki iki bölümlü çizgeler üzerinde DM (Dulmage Mendelsohn) ayrıştırması kullanılarak DM ayrıştırmasının verdiği bilgiler ışığıında çoklama yapılmasına karar verilir.

Bu bölümde Dulmage-Mendelsohn çözümleme metodundan faydalanarak iki bölümden oluşan bir hiperçizgeyi (bipartite hypergraph) kesitte kalan hiperkenar sayısını azaltacak şekilde düğümleri iki yönlü olarak çoklama yöntemleri anlatılacaktır. Şekil 11, örneğin bir bölümlededeki
Vᵢ ve Vⱼ böümleri arasındaki kesiti göstermektedir. Şekilden görüleceği gibi kesitte 11 adet hiperkenar (n₁ ... n₁₁), Vᵢ bölümünde 12 adet sınır düğüm (t₁ ... t₁₂) ve Vⱼ bölümünde 10 adet sınır düğüm (t₁₃ ... t₂₂) bulunmaktadır. Bu kesiti en aza indirmek için geliştirilmiş yaklaşımımız iki sahada oluşmaktadır. Birinci sahadaki amaç, Vᵢ bölümündeki sınır düğümlerden en az düğümü Vⱼ bölümünde çoklayarak kesitten en fazla sayıda hiperkenarı kurtarken. İkinci sahadaki amaç ise birinci saha sonunda kesitten kurtulan hiperkenarlar göz ardı edilerek, Vⱼ bölümündeki sınır düğümlerden en az düğümü Vᵢ bölümünde çoklayarak kesitten en fazla sayıda hiperkenarı kurtarmaktır. Bu iki sahanın sıralaması çözüm kalitesini değiştirebileceği için iki olası sıralama da denen en iyisi uygulanmaktadır.

Şekil 12: Şekil 11'deki kesitler için oluşturulan iki kümeli çizge.

DM ayrışımının Ysol ile Ysağ düğüm sayısı farkını maksimize ettiği göz önüne alınırsa, birim düğüm ağırlığı varsayımlar altında, Ysol'daki düğümlerin Vj bölümüne çoklanması, en az sayıda düğüm çoklaması ile en çok sayıda hiperkenarın kesitinden kurtarılması amacı için kullanılabilir. Şekil 12'de Yatay = (Ysol, Ysağ) ayrımı (\{t_1, t_2, t_3\}, \{n_1, n_2, n_3, n_4, n_5\}) düğümlerinden oluşmaktadır. Bu yatay DM ayrımı, Şekil 1'de Vj bölümündeki t_1, t_2, t_3 sınır düğümlerinin Vj düğümünde çoklanması sonucu n_1, n_2, n_3, n_4, n_5 hiperkenarların kesitinden kurtulacağını göstermektedir. Şekil 13, belirlenen düğümlerin çoklanmasından sonra oluşan ikinci düğüm çoklanması sahasını göstermektedir.

Şekil 13: Şekil 12'de belirlenen düğümlerin çoklanması sonucu oluşan ikinci düğüm çoklanması sahası.

**Şekil 14: DM çözümlemesi kullanarak belli parçaların bulunmasının aşamaları.**

4.2 K Parçalı Bir Hiperçizge Bölümlemesi için Dengeyi Sağlayan Min-Kesit Çoklama Kümesi
Bu çalışmada, verilen bir $H = (V, N)$ hiperçizgesinin $K$ parçalı $\Pi = (V_1, \ldots, V_K)$ bölümlemesindeki kesiti en aza indirgenecek bir çoklama kümesi (replication set) elde etmeye çalışıyoruz. Bu çoklama kümesi $R = (R_1, \ldots, R_K)$ ve $R_k \subseteq V$ ile gösteriliyor olup, elde edilen yeni çoklamalı bölümme kümesi şu şekilde oluşurur: $\Pi^r = (V_1 \cup R_2, \ldots, V_K \cup R_K)$. Dolayısı ile $R$ kümesindeki her bir $R_k, V_k$ parçasına yapılacak düğüm çoklamalarını ifade edmekte. Bu durumda $R$ kümesi böyle oluşturulmalıdır ki, çoklamaların esnasında verilen maksimum çoklama kapasitesi $p$ ve ilk bölümlemenin sahip olduğu denge (imbalance) $X(\Pi)$ oranlarına sadık kalmalıdır. Çoklama kümelerinin hesaplanmasında her parçayı (part) belirli bir sıraya ayrı ayrı ele alarak, şu an işlemekte olduğumuz $V_k$ parçasına komşu (adjacent) olan $Adj(V_k)$ düğümleri ve bu parçadaki düğümleri sağlayan $N_{ext}(V_k)$ kesit hiperkenarlarını göz önüne alıyoruz. Ve her seferinde verilen $X(.)$ kesit boyutu (cutsizes) metriğine bağlı olarak, $Adj(V_k)$ düğümleri ve $N_{ext}(V_k)$ hiperkenarlarından türetilmiş bir $H_k$ alt hiperçizgesi oluşturulur, $V_k$ parçasına yapılacak düğüm çoklamalarını bu hiperçizgeden seçiyoruz. Dolayısı ile sırayla oluşturduğumuz her $H_k$ hiperçizgesi, $V_k$ parçasına yapılacak çoklamalarını temsil eder hale geliyor. $H_k$ hiperçizgesi üzerinde yapılacak düğüm seçimini önceden belirlenmiş bir $\kappa_k$ çoklama kapasitesi sınırlaması altında gerçekleştiriyoruz. Bu noktada $\kappa_k$ böyle seçilmiştir ki, işlem sonunda hem verilen çoklama kapasitesi aşılamanış olmalı, hem de meydana gelen yeni $\Pi^r$ bölümlemesindeki $X(\Pi^r)$ eski $X(\Pi)$ değerinden küçük ya da bu değere eşit olmalı. $H_k$ üzerindeki düğüm seçimlerini yapmak için bir integer linear programming (ILP) (tamsayı doğrusal programlama) formülasyonu kullanıyoruz. Ek olarak, ILP fazındaki iş yükünü azaltmak adına, $H_k$ hiperçizgesini Dulmage-Mendelsohn çözümlemesini baz alan etkili bir algoritma ile küçütterek (coarsening) $H_k^{coarse}$ hiperçizgesini oluşturuyoruz. Daha sonra ILP fazını $H_k$ yerine, $H_k^{coarse}$ üzerinde çalıştırıyoruz. Bahsi geçen bu çoklama algoritmaları genel hatları ile Algoritma 6 üzerinde gösterilmektedir.
Algoritma 6: Çoklama kümesinin bulunması (FIND_REPLICATION_SET(H, Π, W, ρ))

1. π₀₁ ← Π
2. for k ← 1 to K do
3.  \( r_k = (1 + \rho)W_{avg} - w(V_k) \)
4.  \( H_k ← CONSTRUCT(H, k, \pi_{k-1}) \)
5.  \( H^coarse_k ← COARSEN(H_k) \)
6.  \( R_k ← SELECT(H^coarse_k, r_k) \)
7.  \( \pi_k ← \{V_k \cup R_1, \ldots, V_k \cup R_k, V_{k+1}, \ldots, V_K \} \)
8.  UPDATE(k)
9. end for
10. Π₀ ← Π₀

Algoritma 7: Çoklama işleminden sonra yapılan güncelleme algoritması (UPDATE(K))

1. for l ← (k + 1) to K do
2.   for each net \( n_j \in N_{ext}(V_k) \) do
3.     if \( (\text{Pin}(n_j) \cap V_l) \cap R_k \neq \emptyset \) then
4.       for each vertex \( v \in (\text{Pin}(n_j) \cap V_k) \) do
5.         if \( Nets(v) \cap N_{ext}(V_l) = \{n_j\} \) then
6.           \( \text{Adj}(V_l) = \text{Adj}(V_l) - \{v\} \)
7.         end if
8.       end for
9.   \( N_{ext}(V_l) = N_{ext}(V_l) - \{n_j\} \)
10.  \( \Lambda(n_j) = \Lambda(n_j) - V_l \{\text{cut-net metriği için isteğe bağlı}\} \)
11. end if
12. end for
13. end for

Her çoklama sonucu ilişkili \( Adj(.) \) ve \( N_{ext}(.) \) kümeleri de dolaylı ya da doğrudan bu çoklama işlemlerinden etkilenmektedir. Şöyle ki, \( H_k \) üzerinde \( n_j \) hiperkenarının \( V_l \) parçasına ait olan düğümleri seçilmiş ve \( V_k \) parçasına kopyalanmış ise, \( n_j \) hiperkenarı \( N_{ext}(V_l) \) kümesinden kaldırılmıştır. Ya da, \( v_i \) düğümü \( Adj(V_l) \) kümesinde sadece \( n_j \) hiperkenarına bağlı olduğu için duruyorsa ve \( n_j \) hiperkenarını \( N_{ext}(V_l) \) kümesinden kaldırılmışsa, \( v_i \) düğümü de \( Adj(V_l) \) kümesinden kaldırmalıdır. Bu sebeple, her çoklama işlemi sonunda bu değerlerin güncellenmesi için UPDATE(k) çağrılmaktadır. UPDATE(k) işlevi Algoritma 7'de gösterilmiştir.

4.3 \( H_k \) Hiperçizgelerinin İnşası
\( H_k \) hiperçizgeleri üzerinde yapılan düğüm seçimlerinin, \( V_k \) parçasına yapılacak düşün çoklamalarını temsil ettiği belirtmiştir. Bu sebeple, \( H_k \) hiperçizgeleri öyle inşa edilmedi ki, bunlar üzerinde yapılacak düşün çoklamaları ile \( H_k \) üzerindeki kesitten kaldırlacak olan
hiperkenarların getirdiği kesit boyutındaki azalma, aynı şekilde baştaki $\mathcal{H}$ hiperçizgesinde de paralel olarak benzer miktarda kesit boyutunun azalmasına sebep olmalıdır. Bu sebeple, iki farklı kesit boyutu (cut-size ve connectivity) metriği için iki farklı $H_k$ hiperçizgesi ($H^\text{cut}_k$ ve $H^\text{con}_k$) oluşturulmaktadır. Bu sayede $H^\text{cut}_k$ üzerinde yapılan düğüm seçimleri ile kesitten kaldırılan hiperkenarlar, $H_k$ üzerindeki cut-net kesit boyutunda da paralel bir azalmaya neden olacaktır. Benzer şekilde, $H^\text{con}_k$ üzerinde yapılan düğüm seçimleri ile kesitten kaldırılan hiperkenarlar, $H_k$ üzerindeki connectivity kesit boyutunda da paralel bir azalmaya neden olacaktır. $H^\text{cut}_k$ ve $H^\text{con}_k$ hiperçizgelerinin nasıl oluşturulduğu Algoritma 8 ve Algoritma 9 üzerinde gösterilmiştir.

\begin{verbatim}
1: $V^\text{ext}_k \leftarrow \text{Adj}(V_k)
2: \mathcal{N}^\text{cut}_k \leftarrow \mathcal{N}^\text{ext}_k(V_k)
3: \text{for each net } n_j \in \mathcal{N}^\text{ext}_k(V_k) \text{ do}
4: \quad \text{Pins}(n_j) \leftarrow \text{Pins}(n_j) \setminus V_k
5: \text{end for}
6: \text{return } H^\text{cut}_k \leftarrow (V^\text{cut}_k, \mathcal{N}^\text{cut}_k)
\end{verbatim}

Algoritma 8: Cut-net metriği için hiperçizge inşası (CONSTRUCT($H, k, \mathcal{P}_k$))

Algoritma 8'de gösterildiği üzere, $H^\text{cut}_k = (V^\text{cut}_k, \mathcal{N}^\text{cut}_k)$ hiperçizgesinin düğüm ve hiperkenar kümleri ile $\text{Adj}(V_k)$ ve $\mathcal{N}^\text{ext}_k(V_k)$ birebir örtüşmektedir. Tek fark, $\mathcal{N}^\text{ext}_k(V_k)$ kümesindeki hiperkenarların düğümlerinden $V_k$ parçasına ait olan düğümlerinin çıkarılmasıdır. Bu sayede, $V^\text{cut}_k$ kümesinden seçilecek düğümler ile $\mathcal{N}^\text{cut}_k$ kümesinde tüm bacakları çoklanan hiperkenarların $V_k$ parçasına çoklanması birebir modellenmiş olacaktır.

\begin{verbatim}
1: $V^\text{con}_k \leftarrow \text{Adj}(V_k)
2: \mathcal{N}^\text{con}_k \leftarrow \emptyset
3: \text{for each net } n_j \in \mathcal{N}^\text{ext}_k(V_k) \text{ do}
4: \quad \text{for each part } V_i \in \mathcal{N}(n_j) \text{ and } V_i \neq V_k \text{ do}
5: \quad \quad \mathcal{N}^\text{con}_k \leftarrow \mathcal{N}^\text{con}_k \cup \{n^i_j\}
6: \quad \quad \text{Pins}(n^i_j) \leftarrow \text{Pins}(n_j) \cap V_i
7: \quad \text{end for}
8: \text{end for}
9: \text{return } H^\text{con}_k \leftarrow (V^\text{con}_k, \mathcal{N}^\text{con}_k)
\end{verbatim}

Algoritma 9: Connectivity metriği için hiperçizge inşası (CONSTRUCT($H, k, \mathcal{P}_k, \mathcal{K}$))

Algoritma 9 üzerinde $H^\text{con}_k$ hiperçizgesinin nasıl oluşturulduğu gösterilmiştir. Burada, $H^\text{cut}_k$ hiperçizgesinden farklı olarak, her hiperkenar için bir bölünme (net splitting) işlemi uygulanmıştır. Bu işlemde, her hiperkenar bağlı olduğu parça sayısı kadar çoğaltılarak, her kopya, o hiperkenarın sadece o parçadaki düğümlerine bağlanmıştır. Bunun sebebi connectivity
kesit metriğine göre, bir $n_j$ hiperkenarının bütün bacaklarına bağlı olan düğümlerindense, sadece bazı parçalardaki düğümlerinin tamamıyla çoğlanmasını (bu sayede kesit metriğinde kullanılan $\lambda(n_j)$ azaltılmış olacak) yeterli olacaktır. Bu yöntem ile, bu çoğaltılan hiperkenarların bir ya da birden fazlasının bağlı olduğu düğümler kopyalanarak, $\tau$ üzerinde, düğümleri kopyalanan hiperkenarların sayısı ile doğru orantıda bir azalma ifade edilmiş olunur.

Şekil 15: 3 parçadan oluşan örnek bir hiperçizge.

Şekil 15 üzerinde 3 parçadan oluşan örnek bir hiperçizge verilmiştir. Bu hiperçizgde toplam 14 kesit hiperkenarı ve bu kesit hiperkenarlarına bağlı toplam 24 sınır düğümü bulunmaktadır. Örnek teşkil etmesi amacı ile burada $V_i$ parçasına yapılacak olan çoğalmalarda kullanılacak $H^\text{Cut}_i$ ve $H^\text{Con}_i$ hiperçizgelerinin nasıl oluşturulacağı gösterilecektir.
Şekil 16: Örnek $H^\text{cut}$ ve $H^\text{con}$ hiperçizgeleri.

Şekil 16a üzerinde, $H^\text{cut}$ hiperçizgesi gösterilmiştir. Burada, $\text{Adj}(V_1)$ ve $N_{\text{ext}}(V_1)$ aynı alınarak, $N_{\text{ext}}(V_1)$ hiperkenarlarının $V_1$ parçasına bağlı olan bacakları kaldırılmıştır. Örneğin, burada $v_5$, $v_6$, $v_7$ ve $v_9$ düğümlerinin çoklanması $n_{12}$ hiperkenarını $H^\text{cut}$ hiperçizgesinde cut-net metriğine göre kesitten kurtaracaktır.

Şekil 16b üzerinde, $H^\text{con}$ hiperçizgesi gösterilmiştir. Burada çoğaltılan hiperkenarlar kırmızı olarak resmedilmiştir. Örneğin $n_7$ hiperkenarı hem $V_2$ hem de $V_3$ parçalarına bağlı olduğundan, $n_7^2$ ve $n_7^3$ olmak üzere iki farklı hiperkenar olarak ifade edilmiştir. (Bunlar şekilde $\gamma^2$ ve $\gamma^3$ ile gösterilmiştir.) $H^\text{cut}$ ve $H^\text{con}$ arasındaki bu ifade farklılığı şu şekilde açıklanabilir: $v_9$, $v_{10}$, $v_{11}$ düğümlerinin çoklanması $H^\text{cut}$ üzerinde sadece $n_9$ hiperkenarını kesitken kurtarırken, $H^\text{con}$ üzerinde $n_9^2$, $n_9^3$, $n_9^4$ hiperkenarlarını kurtarmaktadır. Benzer şekilde birim düğüm ve hiperkenar ağırlığı kullanıldığı varsayacak olursak, $v_9$, $v_{10}$, $v_{11}$ düğümlerinin çoklanması cut-net metriğini göre kesitin boyutunu 1 birim azaltırken, connectivity metriğine göre kesitin boyutunu 3 birim azaltmaktadır.

4.4 $H_k$ Hiperçizgeleri Üzerinde Düğüm Seçimi

$H_k$ hiperçizgeleri üzerinde yapılacak düğüm seçimlerinin $V_k$ parçasına yapılacak düğüm çoklamalarını temsil ettiği, ve bu hiperçizgeler üzerinde elde edilen kesit boyutundaki azalmalarının paralel olarak $H^\mu$ hiperçizgesi üzerinde de kesit boyutunda azalmaya yol açtığını açıklamıştık. Bu bölümde, $H_k$ hiperçizgeleri üzerinde vereceğimiz bir integer linear programming (ILP) formülasyonu ile düğümlerin nasıl seçileceğini anlatacağız.
Verilen bir \( \kappa_k \) kapasitesi altında, \( H_k \) hiperçizgeleri üzerindeki düğümlerin seçilmesi set-union knapsack problemine denk gelmektedir ve bu problem \( \mathcal{NP} \)-hard bir problem olarak sınıflandırılmaktadır. Bu probleme getirdiğimiz ILP formülasyonu aşağıdaki gibidir.

Verilen formülasyondaki \( X(.) \) değerleri bir hiperkenarın bağlı olduğu tüm düğümlerinin seçilip seçilmesmesine göre boolean bir değer almaktadır. Benzer şekilde, \( y(.) \) değerleri bir düğümün seçilip seçilmediğini ifade ediyor olup, yine boolean bir değer almaktaştir. Formülasyondaki (1) ifadesi bağlı olduğu düğümleri çoklanmış hiperkenarların toplam ağırlığını maksimiz etmeye çalışmaktadır. (2) ifadesi bir \( n_j \) hiperkenarının ancak ve ancak bağlı olduğu tüm düğümleri seçildiğinde \( x(n_j) \) değerinin 1 olmasıına izin vermektedir. (3) ise toplam seçilen düğüm ağırlığının verilen \( \kappa_k \) sınırını aşmamasını sağlamaktadır.

ILP formülasyonu \( H_k \) için optimal çoklama kümesini veriyor olsa da, pratikte \( \mathcal{NP} \)-hard bir problemleri çözüldüğünde oldukça fazla zaman almakta. Bu süreci kısaltmak adına aşağıdaki yöntemler uygulanmaktadır.

- \( H_k \) hiperçizgesinde \( \kappa_k \) miktarında çoklama ile tüm bacakları seçilemeyecek olan hiperkenarları ve sadece bu hiperkenarlara bağlı düğümleri \( H_k \) hiperçizgesinden kaldı.

- \( H_k \) hiperçizgesinin çoklama karakteristiklerinin korunarak daha da küçültülüb, ILP fazına \( H_k \) yerine bu yeni küçük hiperçizgenin verilmesi.

- ILP fazının çalışma zamanının kısıtlanması.

### 4.5 \( H_k \) Hiperçizgelerinin Küçültmesi

Bu bölümde, \( H_k \) hiperçizgelerinin Dulmage-Mendelsohn ayırımı ile \( H_{k\text{ coarse}} \) hiperçizgesine nasıl küçültüldüğü üzerinde duracağız. \( H_k \) hiperçizgesinin \( H_{k\text{ coarse}} \) hiperçizgesine küçültülmesindeki önemli olan nokta, \( H_k \) üzerindeki belirli kalitede çoklamaları temsil eden düşüm ve hiperkenar gruplarının, \( H_{k\text{ coarse}} \) üzerinde de benzer şekilde temsil edilmesi olacaktır. Şöyle ki, az sayıda düşüm kullanarak çok sayıda hiperkenarın tüm bacaklarını bağlayabilen gruplar yüksek kaliteli çoklamaları ifade etmektedir; ortalama sayıda düşüm kullanarak ortalama sayıda hiperkenarın tüm bacaklarını bağlayabilen gruplar ortalama kalitedeki çoklamaları ifade etmektedir; çok sayıda düşüm kullanarak az sayıda hiperkenarın tüm bacaklarını bağlayabilen gruplar düşük kalitedeki çoklamaları ifade etmektedir. Önceki raporumuzda coarse-grained ve fine-grained

44
Dulmage-Mendelsohn ayrışımının nasıl işlediğine dair bilgi vermiştik. Bu açıdan bakıldığında, Dulmage-Mendelsohn ayrışımındaki horizontal, square, vertical grupları sırası ile yüksek, ortalama, düşük kaliteli çoklamalara denk gelmektedir.

\( H_k \) hiperçizgesinin \( H_{coarse} \) hiperçizgesine küçültülmesindeki diğer bir önemli nokta, Dulmage-Mendelsohn ayrışımının iki parçalı çizgelerde (bipartite) çalışıyor olması ve dolayısıyla \( H_k \) hiperçizgesinin küçültme işleminde önce iki parçalı bir çizgeye dönüştürülmesinin gerektiğini görmektedir. Bu dönüşüm işlemi şu şekilde yapılabilir: 1) Her bir \( v_i \) düğümü \( v_i \) şeklinde yeni bir düğüm ile ifade edilir. 2) Her bir \( n_j \) hiperkenarı \( v_{n_j} \) şeklinde yeni bir düğüm ile ifade edilir. 3) \( n_j \) hiperkenarının her bir \( v_i \in PIns(n_j) \) bacağı için yeni bir \( (n_j, v_i) \) kenarı eklenir.

\[ \text{Şekil 17: Örnek } H_{con}^1 \text{ hiperçizgesinin DM ayrışımı ile küçültülmesi.} \]

\[ \text{Şekil 17 üzerinde örnek bir } H_{coarse}^1 \text{ hiperçizgesinin Dulmage-Mendelsohn ayrışımı kullanılarak } H_{con}^1 \text{ hiperçizgesine küçültmesi gösterilmiştir.} \]
4.6 Bölümlemeye Ait DeneySEL Sonuçlar

Çift aşamalı çoklamalı hiperçizge bölümlene kodlarınımda ilk aşama PaToH çok seviyeli hiperçizge bölümlene aracı ile verilen hiperçizgenin K parçaya bölümenmesinden oluşur. İlk aşamada hiçbir çoklama yoktur. İkinci aşamada ise K Parçalı Bir Hiperçizge Bölümlemesi için Dengeyi Sağlayan Min-Kesit Replikasyon Kümesi problemi çözülerek çoklama işlemi gerçekleştirilmektedir.

Bu bölümde değişik veri kümeleri ile yapılan deney sonuçları sunulmaktadır. İlk olarak deneylerde kullanılan veri kümelerinin özellikleri açıklanacak, daha sonra deney parametreleri belirtilcek ve ilk aşama sonunda elde edilen bölümlenmelerin özellikleri tartışılacak ve son olarak da elde edilen deney sonuçları sunulacaktır.

Bu bölümde verilen tabloların sütun başlıkları şu şekilde açıklanabilir:

<table>
<thead>
<tr>
<th>Sütun Başlığı</th>
<th>Açıklama</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam hiperkenar sayısı</td>
</tr>
<tr>
<td></td>
<td>Toplam düğüm sayısı</td>
</tr>
<tr>
<td></td>
<td>Toplam bacak sayısı</td>
</tr>
<tr>
<td></td>
<td>Bir hiperkenarın bağlı olduğu ortalama düğüm sayısı</td>
</tr>
<tr>
<td></td>
<td>Bir düğümün bağlı olduğu ortalama hiperkenar sayısı</td>
</tr>
<tr>
<td></td>
<td>Kesit değeri</td>
</tr>
<tr>
<td></td>
<td>Bölümler arasındaki yük dengesizliği yüzdesi (%)</td>
</tr>
<tr>
<td></td>
<td>Kesitte kalan hiperkenar sayısı</td>
</tr>
<tr>
<td></td>
<td>Sınırda olan (kesitte kalan bir hiperkenra bağlı olan) düğüm sayısı</td>
</tr>
<tr>
<td></td>
<td>Kesitteki bir hiperkenarın bağlı olduğu ortalama düğüm sayısı</td>
</tr>
<tr>
<td></td>
<td>Sınırdaki bir düğümün bağlı olduğu ortalama hiperkenar sayısı</td>
</tr>
<tr>
<td></td>
<td>Çoklama sonrası kesit değerindeki azalım yüzdesi</td>
</tr>
<tr>
<td></td>
<td>Çoklama sonrası yük dengesizliği değerindeki azalım yüzdesi</td>
</tr>
<tr>
<td></td>
<td>Hk kesit hiperçizgelerinin sıcak saylarının ortalaması</td>
</tr>
<tr>
<td></td>
<td>Dulmage-Mendelsohn (DM) çözümlemesini baz alan sıkıştırma algoritmasi ile elde edilen Hk\textsuperscript{coarse} hiperçizgelerinin ortalama bacak sayları ile Hk kesit hiperçizgelerinin ortalama bacak saylarının arasındaki farkın yüzdesi.</td>
</tr>
<tr>
<td>ρ</td>
<td>Çöklama yüzdesi</td>
</tr>
</tbody>
</table>

Tablo 7: Sonuçarda kullanılan terimler.
4.6.1 Veri Kümeleri

Tablo 8'de bu veri kümlerinden elde edilen hiperçizgelerin özellikleri belirtilmektedir.

| Veri Kümesi   | |N| | |V| | |P| |d_<sub>ort</sub>(kenar) | d_<sub>ort</sub>(düğüm) |
|---------------|----------|----------|----------|----------|----------------------------|--------------------------|
| CalGovernor   | 30.805   | 92.279   | 3.004.908| 97,5     | 32,6                       |
| Facebook      | 66.568   | 4.618.974| 14.277.456| 214,5   | 3,1                        |
| Wikipedia     | 70.115   | 1.350.762| 43.285.851| 617,4   | 32,0                       |

Tablo 8: Veri kümesi özellikleri.

4.6.2 İlk Aşama: Hiperçizge Bölümleme
Çift aşamalı çoklamalı hiperçizge bölümleme modelimizde ilk aşamada çoklamasız hiperçizge bölümlemesinin yapılması öngörülmektedir. Bu amaçla Tablo 8'de bilgileri verilen hiperçizgeleri PaToH hiperçizge bölümleme aracını kullanarak K=128 ve K=256 parçaya bölündük. Tablo 9'da bu bölümlemelerde ilgili elde ettiğimiz sonuçları görebilirsiniz.

| Veri Kümesi   | K   | X(Γ)   | ibr(Γ) | |N*| | |V*| |d_<sub>ort</sub>(kenar*) | d_<sub>ort</sub>(düğüm*) |
|---------------|-----|--------|--------|----------|----------------------------|--------------------------|
| CalGovernor   | 128 | 201.391| 5,7    | 24.476   | 92.275                     | 119,6                    | 32,6                       |
|               | 256 | 298.223| 5,1    | 27.724   | 9.278                      | 107,4                    | 32,6                       |
| Facebook      | 128 | 324.393| 1,4    | 58.467   | 4.611.479                  | 234,7                    | 3,1                        |
|               | 256 | 415.405| 1,2    | 61.934   | 4.617.453                  | 225,7                    | 3,1                        |
| Wikipedia     | 128 | 1.040.098| 4,2  | 69.117   | 1.350.568                  | 623,5                    | 32,0                       |
|               | 256 | 1.470.241| 4,9  | 69.608   | 1.350.736                  | 620,5                    | 32,0                       |

Tablo 9: İlk aşamadaki hiperçizge bölümleme sonucunda elde edilen bölümlemelerin özellikleri.

4.6.3 İkinci aşama: Çoklama
Çift aşamalı çoklamalı hiperçizge bölümleme modelimizde ikinci aşamada çoklama işlemlerinin yapılabilmesi için Dengeyi Sağlayan Min-Kesit Çoklama Kümesi problemini çözülmektedir. Bu amaçla Tablo 8'de bilgileri verilen hiperçizgeleri PaToH hiperçizge bölümleme aracını kullanarak K=128 ve K=256 parçaya bölündük. Tablo 9'da bu bölümlemelerle ilgili elde ettiğimiz sonuçları görebilirsiniz.
Tablo 10: Çoklama sonuçları.


4.6.4 Sıkıştırma Algoritmalarının Karşılaştırılması

DM çözümlemesi ile elde edilen sıkıştırma sonuçları çok başarılı sonuçlar verse de alternatif sıkıştırma yöntemlerinin de incelemesinde hayal edilmediği düşündüğeyiz. Bu sebeple PaToH çok-seviyeli hiperçizge bölümlene aracının içinde uygulamaya dönüştürülmiş 17 farklı sıkıştırma yöntemi (HCM, PHCM, MANDIS, AVEDIS, CANBERRA, ABS, GCM, SHCM, HCC, HPC, ABSHCC, ABSHPC, CONC, GCC, SHCC, NC, MNC) modifiye edilerek Hₖ sınır hiperçizgelerinin sıkıştırılmasında kullanılmış ve elde edilen hiperçizgeler ILP çözücüye verilerek yapılan sıkıştırmların elde edilen kesitlere etkisi incelenmiştir. Değişik K ve ρ değerleri ile yapılan bu inceleme sonucunda her bir sıkıştırma algoritmasının başarılı olduğu bazı veri kümeleri, K ve ρ değerleri olsa da genel olarak alternatif sıkıştırma yöntemlerinin tamamının oldukça değişken performanslar verdiği gözlenmiştir. Buna karşılık DM-tabanlı sıkıştırma
yöntemimiz hemen her parametre grubunda başarılı sonuçlar vermiş, deneylerin önemli bir yüzde (87.6) de en iyi 3 değerden birisini vermiştir.

5 GERÇEK ZAMANLI PARALEL SORGU İşLEME SİSTEMİ
Bu bölümde, önerdiğimiz tek aşamalı çoklama şemasını ve bunu bazı alararak geliştirdğimiz çoklama aracı rpPaToH'u gerçek zamanlı bir paralel sorgu işleme sistemi üzerinde test ettik. Bu paralel sistem terim-tabanlı bir ters dizin sistemi olup, kullanılmadan önce terimlerin ters dizinlerinin ön işlemlenmesi üzerine dayanmaktadır. Bölümleme işlemi çeşitli yollarla yapılabilmekte olup (lig-usulü (round-robin) gibi), biz deneylerimizde önerdiğimiz üzere PaToH ve rpPaToH'u bu bölümleme işlemi için kullandık. İlk ana aşamadan oluşan bir süreç sonucunda kullanıcılar tarafından gönderilen sorgular verimli ve kısa bir süreç içinde cevaplanabilmektedir.

İlk aşama, varolan bir döküman korpusu üzerinden elde edilmiş sorgu loglarından bir hiperçizge üretilip bu hiperçizgenin PaToH ya da rpPaToH aracılığıyla bölümlenmesinden oluşmaktadır. Bu aşamada hiperçizge üretilirken düğümler ve kenarlar üzerine çeşitli ağırlık ve pahalar atanabilir. Daha sonra da görüleceği üzere atanan ağırlık veya paha değerleri sorgu işleme sürecinin performansını önemli bir şekilde etkilemektedir. Daha önce de bahsetildiği gibi, her bir sorgu bir hiperkenar ile, bu sorgu tarafından istenen terimler ise bu hiperkenara bağlı düğümler ile modellenmiştir. Verilen bölüm sayısı ve diğer parametreler göre bölümleme aracı çalıtırılmaktadır ve buradan elde edilen düğümlerin bölüm bilgisi paralel veri erişimi sisteminde terimlerin hangi dizin sunucularına atanacağına karar verilmesinde kullanılmaktadır. rpPaToH aracılığıyla elde edilen çoklamalı bölümleme sonucunda çoklanan bir düğüm, paralel sorgu işleme sistemindeki düğümlere denk gelen her bir dizin sunucusunda çoklanmaktadır.


Bu iki aşamalı sistemin ana hatları Şekil 18'de görülmektedir.
Şekil 18: İki aşamalı paralel sorgu işleme sisteminin genel hatları. 1. aşamada sentetik bir sorgu dosyası kullanılarak bölümeleme işlemi gerçekleştirilmektedir. Sonraki aşamada ise bu bölümleme bilgisi kullanılarak kölelerin ters listeleri oluşturulmaktadır ve oluşturulmuş sentetik sorgular kullanılarak paralel sorgu işleme sistemi çalıştırılmaktadır.
5.1 Bölümleme
Bu bölümde döküman yığınının sentetik sorgu üretimi, bu sorgu dosyasından hiperçizge üretimi ve bu hiperçizge dosyasının bölümlenmesi, bölümlenme için kullanılan parametreler ve bölümlleme sonuçları hakkında detaylı bilgi verilecektir.

5.1.1 Döküman Yığını Özellikleri ve Sentetik Sorgu Üretimi

Gerçekçi sorgu loglarının bulunmasının zorluğunu göz önünde tutarak sorgular sentetik olarak üretilmiştir. Üretilen sentetik logların gerçekçi sorgu loglarına benzemesi için, sadece belli bir eşliğin üstünde frekansa sahip olan terimler sorgu loglarında geçecek şekilde ayarlanmış ve dökümanlarda daha sık geçen terimlerin sorgularında da daha sık geçmesi sağlanmıştır. Üretilen sorguların uzunluğu 2-4 terimden oluşmaktadır. Tek terimden oluşan sorguların hiperçizge bölümlemesinde yük dengelemesinde kullanılamayacağı göz önünde tutularak bu tip sorgular üretilmemiştir.


5.1.2 Bölümleme Modeli ve Parametreleri
Önerdğımız hiperçizge modelinde düğümler terimleri, hiperkenarlar ise sorguları temsil edecek şekilde düşünülmüş ve bunlar baz alınarak hiperçizgeler üretilmiştir. Her bir düğümün ağırlığı o düğünün frekansına eşit olacak şekilde ayarlanmıştır. Bu şekilde ağırlıkların atanması yapılan disk erişim sayıları üzerinden kesin bir yüksek dengelemesini sağlamakta olup, ters dizin sunucularının disk erişim hacminin azaltılabilmeğini iddia etmektedir. Veri sistemleri isminin ağırlığını iki etmenin ortak olduğunu ve gerçekçiliğini üzerinde de geçiş de olsa bir yüksek dengelemesi sağlanabilmektedir. Bu şekilde, paralel sorguların seçileceği iletişim hacminin azaltılabilmeğini iddia etmektedir ve gerçekçi sonuçlarda da görüleceği üzere bu iddiamız doğrulanmaktadır. Her bir hiperkenarın pahası birim paha olarak atanmıştır. Daha önce de bahsettigimiz üzere sadece sorgu dosyası baz alınarak hiperçizge üretilmiştir. Bunun sebebi, önerilen modelin doğrulanması sırasında yine aynı sorguların paralel sorgu işleme sisteminde
kullanılacak olmasıdır. Tablo 1’de sentetik olarak üretilen 20.000 sorgudan üretilen hiperçizgenin özellikleri verilmiştir:

<table>
<thead>
<tr>
<th>Düğüm</th>
<th>Hiperkenar</th>
<th>Bacak</th>
<th>Max Düğüm</th>
<th>Ort. Düğüm</th>
<th>Max Hiperkenar</th>
<th>Ort. Hiperkenar</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.439</td>
<td>20 000</td>
<td>59.996</td>
<td>389</td>
<td>3,886</td>
<td>4</td>
<td>3,000</td>
</tr>
</tbody>
</table>


Önerdiğimiz modelde yapılan bir düğüm çökaması, paralel sorgu işleme sisteminde yapılacak olan bir ters liste vektörü çökamasına denk gelmektedir. Bu şekilde bir sorgunun işlenebileceği ortalama sunucu sayısını azaltarak, ortaya çıkacak toplam iletişim hacminin düşürülmesi ve sonuç olarak sorguların daha hızlı cevaplanması hedeflenmiştir.

Bölümleme sırasında kullanılan parametreler aşağıdaki belirtildiği gibidir:

- **Bölüm sayısı (K):** 4, 8, 16
- **Dengesizlik katsayısı:** 0.20
- **İlk dengesizlik katsayısı:** 0.24
- **Çoklama değerleri:** 0.10, 0.25
- **Aritma algoritması:** PaToH için BFM, rpPaToH için rFM.
- **Her seviyede arıtma algoritma koşma sayısı:** 2
- **Maksimum izin verilen pozitif olmayan kazanç operasyonu:** 100

Bunlar dışında, hiperçizge bölümleme algoritmalarında genel olarak kullanılan iki çeşit kesit metriği test sonuçlarımızda rapor edilmiştir. Bu iki metrikten ilki olan **CON** metriği (bağlantı metriği) bir bölümleme sonucu elde edilen hiperçizgede her bir hiperkenarın kesite katkısını bağlı olduğu bölüm sayısının bir eksiğinin bu hiperkenara ait paha ile çarpılması olarak hesaplanmaktadır. Diğer metrik olan **CUT** metriği (kesit metriği) ise bir bölümleme sonucu elde edilen hiperçizgede her bir hiperkenarın kesite katkısını o hiperkenarın pahası olarak hesaplamaktadır. Bölümleme sonuçlarında iki metrik de PaToH ve rpPaToH için denenmiş ve elde edilen bölümleme bilgisi repl-ABCServer paralel sorgu işleme sisteminde test edilmiştir.

5.1.3 Bölümele Sonuçları
Bu bölümde 20.000 sorgudan üretilen hiperçizgenin bölümele sonuçları verilmiştir. Bölümleme sonuçları PaToH ve rpPaToH’u içermektedir. rpPaToH için iki farklı çoklama değeri (0.10 - %10 ve 0.25 - %25) kullanılmıştır. Her bir şema CON ve CUT metrikleri için iki farklı sonuç içermektedir.

<table>
<thead>
<tr>
<th>Şema</th>
<th>Kesit Değeri</th>
<th>Dengesizlik Değeri (%)</th>
<th>Çoklama Kullanımı (%)</th>
<th>Gelişme (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaToH-CON</td>
<td>13484</td>
<td>18,028</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>2828</td>
<td>2,695</td>
<td>99,87</td>
<td>79,02</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>232</td>
<td>1,353</td>
<td>99,58</td>
<td>98,27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Şema</th>
<th>Kesit Değeri</th>
<th>Dengesizlik Değeri (%)</th>
<th>Çoklama Kullanımı (%)</th>
<th>Gelişme (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaToH-CUT</td>
<td>10654</td>
<td>16,501</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>2635</td>
<td>2,424</td>
<td>99,87</td>
<td>75,26</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>120</td>
<td>0,385</td>
<td>99,91</td>
<td>98,87</td>
</tr>
</tbody>
</table>

Tablo 12: Bölüm sayısı 4 (K = 4) için çeşitli değerler ve karşılaştırmaları. Gelişme oranları kesit değerleri üstünden olup, PaToH-CON ve PaToH-CUT değerleri baz alınarak hesaplanmıştır.

<table>
<thead>
<tr>
<th>Şema</th>
<th>Kesit Değeri</th>
<th>Dengesizlik Değeri (%)</th>
<th>Çoklama Kullanımı (%)</th>
<th>Gelişme (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaToH-CON</td>
<td>17374</td>
<td>12,594</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>7125</td>
<td>6,699</td>
<td>99,80</td>
<td>58,99</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>2671</td>
<td>5,670</td>
<td>99,92</td>
<td>84,62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Şema</th>
<th>Kesit Değeri</th>
<th>Dengesizlik Değeri (%)</th>
<th>Çoklama Kullanımı (%)</th>
<th>Gelişme (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaToH-CUT</td>
<td>11723</td>
<td>13,354</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>5663</td>
<td>4,086</td>
<td>99,83</td>
<td>51,69</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>2110</td>
<td>6,011</td>
<td>99,76</td>
<td>82,00</td>
</tr>
</tbody>
</table>

Tablo 13: Bölüm sayısı 8 (K = 8) için çeşitli değerler ve karşılaştırmaları. Gelişme oranları kesit değerleri üstünden olup, PaToH-CON ve PaToH-CUT değerleri baz alınarak hesaplanmıştır.
Tablo 14: Bölüm sayısı 16 ($K = 16$) için çeşitli değerler ve karşılaştırmaları. Gelişme oranları kesit değerleri üstünden olup, PaToH-CON ve PaToH-CUT değerleri baz alınarak hesaplanmıştır.

Tablo 12, 13, ve 14’te sırasıyla $K = 4, 8, ve 16$ için bölümleme sonuçları verilmiştir. Tablolarda CON metriği CON metrikleriyle, CUT metriği ise CUT metrikleriyle kıyaslanmıştır. Örneğin Tablo 14’teki ($K = 16$), rpPaToH %25 çoklama miktarı kullanarak CON metriğine göre 6249 kesit değeri elde etmiş ve aynı CON metriğini kullanarak 19792 kesit miktarı elde edilen PaToH’a göre kesit miktarını %68,42 geliştirmiştir. PaToH’ta çoklama miktarı kullanılmadığı için ve gelişme miktarları PaToH baz alınarak hesaplandığı için tabloda o alana denk gelen değerler boş bırakılmıştır.

5.2 Paralel Sorgu İşleme

5.2.1 Paralel Sistem Mimarisi

Gerçek zamanlı olarak çalışan paralel sorgu işleme sistemimiz (repl-ABCServer) yönetici-köle mimarisiinde olup, yönetici merkezi simsal işlemi görev yapar. Kendisine gelen sorgulara göre bu sorguları cevaplayacak olan ters dizin sunucularını belirlemekte ve buna göre sorguların kısmi sorgular yaratıp bunları ilgili sunuculara göndermektedir. Kendilere gelen kısmi sorgulara göre her bir köle disklerinden gerekli terimlerin ters liste dizin girdilerini okuyup verilen şemaya göre bunlardan yeni bir liste yapıp merkezi simsalara yollamaktadır. Burada bahsedilen şema VE (AND) veya VEYA (OR) şeması olabilir. Bu iki sorgu işleme yönteminin ayrıntıları aşağıdaki verilmiştir. Örnek bir sorgu işleme yöntemi Şekil 19'da görülmektedir. Bu sistemde terimler dört ters dizin sunucu arasında paylaşılır. Toplamda 4 terim bulunmaktadır olup, SUNUCU1 t₁ ve t₄, SUNUCU₂ t₂, t₆ ve t₈, SUNUCU₃ t₅, SUNUCU₄ ise t₃ ve t₇ nin ters döküman vektörlerine (TDV) sahiptir. Bu örnek üzerinden anlatacağımız üzere repl-ABCServer’ın sorgu işleme adımlar genel olarak şu şekildeşidir:

- **Adım 1:** Kullanıcının sorgusunun merkezi simsal göndermesi. Sistemimizde kullanıcı simülasyonu yapan bir işlem bulunmaktadır olup, üretilen sentetik sorgulara kendisine verilen parametreye göre bir sıkış miktarı belirleyip buna göre yöneticisi verilmektedir. Şekil 19’daki örnekte görüldüğü üzere merkezi simsal gelen sorgu t₁, t₂, t₄ ve t₇ terimlerini içermektedir.

- **Adım 2:** Merkezi simsal kendisine gelen sorgulardaki terimlere göre, kendisinde bulunan terim-sunucu haritasını kullanarak bu ters dizin sunucularına gönderilmek üzere kısmi sorgular oluşturulmaktadır. Örneğin, SUNUCU₁ e yollanacak kısmi sorgu $<t₁, t₄>$ şeklindedir. Şekil 19’daki örnekte görüldüğü gibi bu sorguya cevap verecek sunucular SUNUCU₁, SUNUCU₂ ve SUNUCU₄ ‘tür.
Şekil 19: repl-ABCServer paralel sorgu işleme sisteminde örnek bir sorgunun terim-tabanlı bölümlenmiş sistemde işlenmesi.

- **Adım 1:** Sorgunun kullanıcı tarafından yöneticiye yollanması
- **Adım 2:** Kısmi sorgu oluşturulması
- **Adım 3:** Kısmi sorguların yollanması
- **Adım 4:** Kısmi sorguların ters döküman vektörlerinin (TDV) diskten erişimi, skorların hesaplanması ve merkezi simsara kısmi cevap listelerinin yollanması
- **Adım 5:** Ters dizin sunucularından gelen kısmi cevap listelerinin birleştirilmesi ve cevapların hazırlanması
- **Adım 6:** Kullanıcıya cevapların yollanması

**Adım 3:** Bu adımda terim-sunucu haritası kullanılarak seçilen sunuculara oluşturulan kısmi sorgular yollanmaktadır.

**Adım 4:** Ters dizin sunucuları kendilerine gelen kısmi sorguların terimlerine göre disk erişimi yapıp gerekli ters döküman vektörlerini ana belleğe almaktadır. Daha sonra belirlenen sorgu cevaplama yöntemine göre (VE, VEYA) bu vektörleri birleştirirken, skorlarını birbirlerine eklemekte, eğer gerekirse bu listeyi küçültmekte ve merkezi simsara gönderilmek üzere sıralayıp kısmi cevap listeleri oluşturmakta ve son olarak bunları merkezi simsara yollamaktadır.

**Adım 5:** Merkezi simsar kendisine gelen kısmi cevap listelerini kullanarak bunları birleştirirken ve kullanıcının istediği sayıda dökümanı ona geri yollamaktadır. Kullanıcıya yollanan dökümanlar en yüksek skorlu dökümanlar olmakta olup merkezi simsar kısmi cevap listelerini birleştirirken bu işi de halletmiş olmaktadır.

**Adım 6:** Merkezi simsar tarafından hazırlanan en yüksek skorlu dökümanlar kullanıcıya geri gönderilmiştir.
Şekil 19'da görüleceği üzere verilen bu örneğe çoklama dahil edilmemiştir. Çoklama düşünüleceği zaman farklı zamanlama algoritmaları kullanılması gerekmektedir çünkü bir terim birden fazla sunucuda bulunabilemektedir. Bu zamanlama algoritmalarını ve detaylarını ilerleyen bölümlerde göreceğiz.

5.2.2 İki Farklı Sorgu Şeması

Daha önce de bahsettğimiz gibi sorgu işleme yöntemleri iki farklı kategoriye ayrılabilir: VE ve VEYA yöntemi. Bu iki yöntemin temel farkı, VE yönteminin verilen sorgudaki tüm terimlerin tamaminin geçtiği dökümanları skorlarına göre kullanıcının döndürüken VEYA yönteminin bu terimlerden en az birinin geçtiği dökümanları o skorlarına göre kullanıcının döndürmesidir. Bu bölümde bu iki yöntemin merkezi simsar ve ters dizin sunucuları açısından farklılıklarını açıklamaya çalışacağız.

Şekil 20: Sorgu işleme sırasında kullanılan VE ve VEYA yöntemlerinin ters dizin sunucuları açısından farklılıklarını.

Şekil 20 bu iki farklı yöntemin ters dizin sunucuları açısından temel bir farklılığını ortaya koymaktadır. Şekilde görüldüğü üzere, kendisine gelen kısımlı sorgu sözcüğünün içerdiği terimlerin listelerini birleştirirken parametrık olan verilen sorgu işleme yönteminin farklılığına göre farklı birlestirmeler yapmaktadır. VE yönteminde, terimlerin geçtiği bütün ortak dökümanlar bulunurken (örnekte t₁ ve t₄'ün geçtiği ortak dökümanlar d₄ ve d₁₀'dur) VEYA yönteminde...
terimlerin geçtiği ortak olsun olmasın bütün dökümanlar alınmaktadır (örnekte t₁ ve t₄’ün geçtiği tüm dökümanlar d₁, d₄, d₅, d₉ ve d₁₀’dur). Bu dökümanların skorları daha sonra gereksinme birbirine eklenip skorlarına göre büyükten küçüğe doğru sıralanmaktadır. Burada dikkat edilmesi gereken en önemli nokta VE yönteminde (i) hesaplanan kısmi cevap listesinin hepsinin yollanması gerektiğini ve (ii) eğer bir ters dizin sunucusuna sadece bir terim içeren bir kısmi sorgu gelirse gönderilen listenin çok büyük olabileceğini. VEYA yönteminde hesaplanan kısmi listelerin hepsinin yollanmasına gerek olmamakla birlikte, döküman skorlarına göre sıralanmış listenin sadece %1’lik bir kısmının sunucular tarafından merkezi simsara gönderilmesi kullanıcıya gönderilen sonuçların kalitesinde hemen hiçbir bozukluğa yol açmamaktadır.

Şekil 21: Merkezi simsarin kendisine gelen kısmi cevap listelerini (KCL) iki farklı yöntemde birleştirmesi.

Şekil 21’de VE ve VEYA yöntemlerinin merkezi simsara açıklıkta farklılıklarını görmekteyiz. Bu örnekte, merkezi simsara gelen soruyu SUNUCU₁ ve SUNUCU₂ cevaplamış olup merkezi simsara gönderdikleri kısmi cevap listeleri (KCL) örnekte görülüğü üzere KCL₁ = <d₁ d₅ d₇> ve KCL₂ = <d₄ d₅ d₇ d₉> şeklindedir. VE yönteminde merkezi simsara düşen görev sadece skorları
eklemekle kalmayıp ayrıca kendisine gelen listelerdeki ortak dökümanları seçmektir. Bu sebeple ters dizin sunucuları ellenindeki kısmi çevap listelerindeki tüm girdileri yollamak durumundadırlar çünkü düşük skor sebebiyle yollanmayan bir döküman-skor ikilisi kullanıcıya gönderilmesi gereken bir cevabin yollanamasına sebep verebilir. Özellikle tek terimli kısımlar bu yüzden yüksek komunikasyon hacmine sebep olmaktadır. VEYA yönteminde ise böyle bir durum yoktur ve bir ters dizin sunucusu hazırladığı döküman skorlarına göre sıralanmış bir kısımlı listenin büyüklüğünün %1’ini yollasa da sonuçların doğruluğunda ve kalitesinde pek bir değişiklik olmamaktadır. Bu yüzden VEYA yönteminin daha az iletişim hacmine sebep vermesi beklenmektedir.

5.2.3 Zamanlama (Scheduling) Algoritmaları
Bu bölümde çoklanın sebep olduğu sorguda geçen çoklanmış bir terimin seçiminin indirgendiği problemden ve bunların seçiminin kullanılan ikili algoritmada bahsedeceğiz.

5.2.3.1 Küme Kapama Problemi ve Algoritmaları
Bu problemde önceki bölümlerde bahsetmiş olup, orada önderliğimiz algoritmaları burada kullanmaktayız.

5.2.3.2 Dinamik Yük Dengelemesi
Bu yöntem sadece ikinci aşamada paralel sorgu işleme sisteminin koşulmasında kullanılabilmesidir çünkü sistemin dinamik bilgilerine göre çoklanan terimler seçilmiştir. Bu yöntemdeki ana hedef dinamik olarak her bir ters dizin sunucusuna atanan yükü dengelemektir. Bunu sağlamak için iki farklı bilgi kullanmaktayız. Öncelik bilgilerden ilki verilen herhangi bir zamanda ters dizin sunucularında işlenme bekleyen sorgu sayısıdır. İkincisi ise her bir ters dizin sunucusunun verilen kadar ne kadar komunikasyon yaptıgıdır. İkinci bilgi gereklilik durumunda bu eşittili kırıktır için kullanılmaktadır. Bu zamanlama sisteminde bir sorgunun hangi sunuculara atanana belirlenirken, her bir sunucunun dinamik olarak ne kadar sorgu sahip olduğu göz önüne alınmaka, hangi sunucuda en az sorgu var ise o sunucular seçilmektedir. Bir eşittilik durumunda ise eşitliğe dahil olan sunucular arasında şu ana kadar hangi sunucu en az komunikasyona sebep verdiği sunucu seçilmiştir.

5.2.4 Sistem Özellikleri ve Parametreleri
Kodlama C dilinde yapılmış olup paralel sistemde mesaj gönderimi için MPI’in LAM/MPI 7.0.6 uyarlaması kullanılmıştır. Daha önce de bahsettiğimiz üzere paralel sorgu işleme mimarisi yönetici-köle mimarısındadır. Her bir bilgisayarda Intel Pentium 4 3.00GHz işlemci, 2GB bellek ve 1KB cache bulunmaktadır.

Paralel sorgu işleme sisteminde her bir ters dizin sunucusu belleğinde ters döküman vektörlerine ait bir dizin bulundurmakta ve bu vektörlerin kendilerini diskte saklamaktadır. VE
yönteminde sunucular kısmi cevapların hepsini merkezi simsara yollamakta, VEYA yönteminde ise tüm döküman yiğinin %1’lik kısmını yollamaktadırlar. Sistemde 20.000 sorgu koşarken, bunu 30.000’e çıkarıp ilk ve son 5.000 sorguyu istatistiklere dahil etmedik. Bu sorguları başka bir bilgisayarın işletim sistemi işlemleri tarafından simüle edilen 50 kullanıcı merkezi simsara yollamaktadır. Kullanıcıya en yüksek skorlu 1000 döküman geri döndürülmektedir.

Paralel sorgu işleme sisteminin 4, 8 ve 16 bölüm için çalıştırdık. Sistemimizde round-robin, PaToH ve rpPaToH için sonuçlar aldık. PaToH ve rpPaToH için iki farklı kesit metriğini (CON ve CUT) test ettik. Buna ek olarak rpPaToH için iki farklı çoklama değeri (0.10 ve 0.25) kullandık. Test sonuçlarımızın hepsi VE ve VEYA yöntemlerinin ikisi içindealdık. Sonuçlar aynı zamanda iki farklı zamanlama algoritmasını da içermektedir.

5.2.5 Sorgu Performans Sonuçları

<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>5.360.853</td>
<td>10,4</td>
<td>9481508</td>
<td>11,1</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>4.061.266</td>
<td>63,6</td>
<td>9482080</td>
<td>70,0</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>3.681.395</td>
<td>67,7</td>
<td>9502016</td>
<td>62,9</td>
</tr>
<tr>
<td>rrPaToH-CON (%10)</td>
<td>365.630</td>
<td>0,8</td>
<td>9496619</td>
<td>15,9</td>
</tr>
<tr>
<td>rrPaToH-CUT (%10)</td>
<td>366.309</td>
<td>0,4</td>
<td>9503257</td>
<td>4,5</td>
</tr>
<tr>
<td>rrPaToH-CON (%25)</td>
<td>199.629</td>
<td>2,1</td>
<td>9492645</td>
<td>4,2</td>
</tr>
<tr>
<td>rrPaToH-CUT (%25)</td>
<td>194.821</td>
<td>0,7</td>
<td>9482524</td>
<td>4,2</td>
</tr>
</tbody>
</table>

Tablo 15: K = 4, Küme Örte Alg, VE
<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>692.067</td>
<td>1,5</td>
<td>9484536</td>
<td>11,0</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>587.221</td>
<td>23,7</td>
<td>9501707</td>
<td>62,7</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>599.291</td>
<td>23,3</td>
<td>9477768</td>
<td>69,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>400.769</td>
<td>0,3</td>
<td>9492950</td>
<td>11,2</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>401.823</td>
<td>0,5</td>
<td>9494340</td>
<td>7,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>358.041</td>
<td>0,5</td>
<td>9502976</td>
<td>5,4</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>356.224</td>
<td>0,7</td>
<td>9488090</td>
<td>4,6</td>
</tr>
</tbody>
</table>

Tablo 16: K = 4, Külme Örtme Alg, VEYA

<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>7.264.130</td>
<td>24,3</td>
<td>9493449</td>
<td>26,3</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>5.007.194</td>
<td>77,8</td>
<td>9518072</td>
<td>68,3</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>5.676.645</td>
<td>104,4</td>
<td>9496372</td>
<td>97,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>850.076</td>
<td>1,2</td>
<td>9502685</td>
<td>16,6</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>874.710</td>
<td>1,1</td>
<td>9487370</td>
<td>11,3</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>367.554</td>
<td>1,8</td>
<td>9498743</td>
<td>5,9</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>359.022</td>
<td>1,2</td>
<td>9487037</td>
<td>5,7</td>
</tr>
</tbody>
</table>

Tablo 17: K = 8, Külme Örtme Alg, VE
<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>779.445</td>
<td>2,5</td>
<td>9485589</td>
<td>25,3</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>650.486</td>
<td>17,3</td>
<td>9503083</td>
<td>68,0</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>674.321</td>
<td>19,3</td>
<td>9497272</td>
<td>98,1</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>473.861</td>
<td>37,2</td>
<td>9496335</td>
<td>21,9</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>472.596</td>
<td>0,5</td>
<td>9479876</td>
<td>11,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>398.567</td>
<td>1,1</td>
<td>9483028</td>
<td>8,6</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>394.906</td>
<td>0,4</td>
<td>9516540</td>
<td>6,8</td>
</tr>
</tbody>
</table>

Tablo 18: $K = 8$, Küme Örtme Alg, VEYA

<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>8.365.334</td>
<td>68,8</td>
<td>9494746</td>
<td>69,4</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>5.970.039</td>
<td>134,9</td>
<td>9504370</td>
<td>118,9</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>6.841.190</td>
<td>102,8</td>
<td>9524168</td>
<td>98,4</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>1.488.321</td>
<td>2,8</td>
<td>9495914</td>
<td>16,6</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>1.677.345</td>
<td>0,9</td>
<td>9476088</td>
<td>19,0</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>752.079</td>
<td>6,3</td>
<td>9491217</td>
<td>15,0</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>717.732</td>
<td>2,4</td>
<td>9490395</td>
<td>21,1</td>
</tr>
</tbody>
</table>

Tablo 19: $K = 16$, Küme Örtme Alg, VE
<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>825.674</td>
<td>9,8</td>
<td>9499064</td>
<td>68,9</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>697.391</td>
<td>26,7</td>
<td>9487730</td>
<td>119,2</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>725.712</td>
<td>18,5</td>
<td>9509597</td>
<td>98,1</td>
</tr>
<tr>
<td>rpPaToH-CON</td>
<td>537.226</td>
<td>0,9</td>
<td>9504249</td>
<td>21,6</td>
</tr>
<tr>
<td>rpPaToH-CUT</td>
<td>546.444</td>
<td>1,5</td>
<td>9491824</td>
<td>25,3</td>
</tr>
<tr>
<td>rpPaToH-CON</td>
<td>457.128</td>
<td>1,8</td>
<td>9480144</td>
<td>13,6</td>
</tr>
<tr>
<td>rpPaToH-CUT</td>
<td>450.008</td>
<td>4,5</td>
<td>9489456</td>
<td>11,4</td>
</tr>
</tbody>
</table>

Tablo 20: K = 16, Küme Örtme Alg, VEYA

<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>5.360.853</td>
<td>10,4</td>
<td>9481508</td>
<td>11,1</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>3.681.395</td>
<td>67,7</td>
<td>9502016</td>
<td>62,9</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>4.061.266</td>
<td>63,6</td>
<td>9482080</td>
<td>70,0</td>
</tr>
<tr>
<td>rpPaToH-CON</td>
<td>2.989.120</td>
<td>3,9</td>
<td>9485256</td>
<td>5,2</td>
</tr>
<tr>
<td>rpPaToH-CUT</td>
<td>3.040.094</td>
<td>1,6</td>
<td>9489118</td>
<td>2,7</td>
</tr>
<tr>
<td>rpPaToH-CON</td>
<td>2.275.233</td>
<td>4,1</td>
<td>9490765</td>
<td>5,1</td>
</tr>
<tr>
<td>rpPaToH-CUT</td>
<td>2.232.509</td>
<td>2,3</td>
<td>9469318</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Tablo 21: K = 4, Dinamik Yük Dengeleme Alg, VE
<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>692.067</td>
<td>1,5</td>
<td>9484536</td>
<td>11,0</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>587.221</td>
<td>23,7</td>
<td>9501707</td>
<td>62,7</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>599.291</td>
<td>23,3</td>
<td>9477768</td>
<td>69,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>555.217</td>
<td>3,5</td>
<td>9469044</td>
<td>4,9</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>559.565</td>
<td>2,7</td>
<td>9482295</td>
<td>3,8</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>518.566</td>
<td>1,2</td>
<td>9476571</td>
<td>3,7</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>519.009</td>
<td>1,8</td>
<td>9482391</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Tablo 22: K = 4, Dinamik Yük Dengeleme Alg, VEY

<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>7.264.130</td>
<td>24,3</td>
<td>9493449</td>
<td>26,3</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>5.007.194</td>
<td>77,8</td>
<td>9518072</td>
<td>68,3</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>5.676.645</td>
<td>104,4</td>
<td>9496372</td>
<td>97,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%10)</td>
<td>4.867.803</td>
<td>4,2</td>
<td>9494395</td>
<td>3,5</td>
</tr>
<tr>
<td>rpPaToH-CUT (%10)</td>
<td>4.970.742</td>
<td>4,5</td>
<td>9488057</td>
<td>3,9</td>
</tr>
<tr>
<td>rpPaToH-CON (%25)</td>
<td>3.980.869</td>
<td>2,0</td>
<td>9481393</td>
<td>5,4</td>
</tr>
<tr>
<td>rpPaToH-CUT (%25)</td>
<td>4.103.706</td>
<td>3,0</td>
<td>9481575</td>
<td>5,9</td>
</tr>
</tbody>
</table>

Tablo 23: K = 8, Dinamik Yük Dengeleme Alg, VE
Tablo 2: K = 8, Dinamik Yük Dengeleme Alg, VEYA

<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>779.445</td>
<td>2,5</td>
<td>9485589</td>
<td>25,3</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>650.486</td>
<td>17,3</td>
<td>9503083</td>
<td>68,0</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>674.321</td>
<td>19,3</td>
<td>9497272</td>
<td>98,1</td>
</tr>
<tr>
<td>rpPaToH-CON</td>
<td>645.129</td>
<td>11,1</td>
<td>9474916</td>
<td>12,4</td>
</tr>
<tr>
<td>(%10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rpPaToH-CUT</td>
<td>656.541</td>
<td>2,9</td>
<td>9485928</td>
<td>6,5</td>
</tr>
<tr>
<td>(%25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rpPaToH-CON</td>
<td>618.132</td>
<td>2,3</td>
<td>9472292</td>
<td>5,1</td>
</tr>
<tr>
<td>(%25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rpPaToH-CUT</td>
<td>624.327</td>
<td>2,4</td>
<td>9474692</td>
<td>4,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tablo 5: K = 16, Dinamik Yük Dengeleme Alg, VE
<table>
<thead>
<tr>
<th>Şema</th>
<th>İletişim Hacmi</th>
<th>Disk Hacmi</th>
<th>Disk Erişimi</th>
<th>Diğer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
<td>Toplam (x10^3)</td>
<td>Deng.</td>
</tr>
<tr>
<td>RR</td>
<td>825.674</td>
<td>9,8</td>
<td>9499064</td>
<td>68,9</td>
</tr>
<tr>
<td>PaToH-CON</td>
<td>697.391</td>
<td>26,7</td>
<td>9487730</td>
<td>119,2</td>
</tr>
<tr>
<td>PaToH-CUT</td>
<td>725.712</td>
<td>18,5</td>
<td>9509597</td>
<td>98,1</td>
</tr>
<tr>
<td>rpPaToH-CON  (%10)</td>
<td>711.776</td>
<td>3,9</td>
<td>9475244</td>
<td>9,7</td>
</tr>
<tr>
<td>rpPaToH-CUT  (%10)</td>
<td>723.661</td>
<td>3,2</td>
<td>9493409</td>
<td>12,6</td>
</tr>
<tr>
<td>rpPaToH-CON  (%25)</td>
<td>696.298</td>
<td>3,0</td>
<td>9487637</td>
<td>12,0</td>
</tr>
<tr>
<td>rpPaToH-CUT  (%25)</td>
<td>704.402</td>
<td>3,3</td>
<td>9488660</td>
<td>10,0</td>
</tr>
</tbody>
</table>

Tablo 26: K = 16, Dinamik Yük Dengeleme Algı, VEYA

Yukarıda verilen tablolarda kullanılan değişkenlerin açıklımı şu şekildedir:

- **İletişim Hacmi**: Merkezi simsar ve ters dizin sunucuları arasındaki mesajların toplam hacmi (bir float cinsinden skor ve bir de int cinsinden doküman birim miktar olarak alınmıştır). Deng sütunu sunucuların yaptığı kominikasyon hacimleri arasındaki dengesizliği göstermektedir olup % cinsinden verilmiştir.

- **Disk Hacmi**: Diskten okunan toplam veri miktarı (miktarın cinsi kominikasyon ile aynıdır). Deng sütunu sunucuların yaptığı disk hacimleri arasındaki dengesizliği göstermektedir.

- **Disk Erişimi**: Toplam yapılan disk erişimi ve sunucular arasındaki dengesizlik.

- **Ort Cevap**: Bir sorguya cevap verilme süresi, saniye cinsinden.

- **Çıktı**: Saniyede kaç tane sorgu cevaplandığı.

Tablo 15-20’de bahsettiklerimiz küme örtme algoritması, Tablo 21-26’da ise dinamik yük dengeleme algoritması kullanılmıştır. Tablo 15-16 (ve bundan sonraki her ikili tablo) sırasıyla VE ve VEYA sorgu işleme yöntemlerini içermektedir. Bölüm sayısı (K) 4, 8 ve 16 için sonuçlar alınmıştır. Tablolarda round-robin lig usulü dağıtımı, PaToH-CON ve PaToH-CUT sırasıyla CON
ve CUT metriklerine göre bölülenmiş hiperçizgelerden üretilen bilgileri kullanarak yapılan sorgu işlemevi, benzer şekilde rpPaToH-CON ve rpPaToH-CUT da sırasıyla CON ve CUT metriklerine göre yapılan bölümeleme sonucunda çıkan bilgiyi kullanarak yapılan sorgu işlemevi göstermektedir. rpPaToH için parantez içindeki değerler çoklama miktarları olup daha önce de bahsedildiği üzere %10 ve %25 çoklama miktarları için sonuçlar alınmıştır. Dikkat edilmesi gereken nokta ise, round-robin, PaToH-CON ve PaToH-CUT için verilen zamanlama algoritmalarının bir farklı olmadağı için sonuçlar bir daha alınmamış ve tablolarda (örneğin Tablo 15 ve 21, Tablo 16 ve 22 vs.) bu değerlerle gelen değerler görüldüğü üzerine aynıdır.

Tablolardaki verilere bakılarken şu sonuçlar çıkarılabilir:

- Çoklama her zaman işe yaramaktadır ve ortalama cevap süresini ve çıktı sayısını büyük miktarda azaltabilmektedir.
- Çoklamasız bir bölümeleme (PaToH), yüksek dengeleme problemleri yaşamaktadır. Bu yüzden iletişim hacmin ne kadar düşse de, sunuculardan bir ya da birkaç tıkanıklık oluşturur ve oritalama cevap süresi ve çıktı miktarları round-robin’e göre çok iyi olmamaktadır. Round-robin yüksek dengeleme iyi sağlayan bir dağıtım yöntemi olarak bilinmektedir. Burada PaToH, daha az iletişim hacmine sahip olmasına rağmen iyi bir yüksek dengeleme yapamadığı için pek bir gelişme gösterememektedir. Çoklama, yukarıda
bahsedildiği üzere, bu iki soruna da çare üretbilmekte, hem iletişim hacminin azaltırken hem de çok iyi bir şekilde yük dengeleme sağlayabilmektedir.

6 SONUÇ

Bu proje, hiperçizge bölümlene de tek ve çift aşamalı düğüm çoklama yöntemleri önermekte ve bu yöntemlerin hiperçizge bölümlene yönteminin etkin biçimde kullanıldığını paralel sorgu işleme problemi üzerinde uygulamalarını incelemektedir. Tek aşamalı yöntem iki bölümlü ve FM-tabanlı bir çoklama algoritmasına dayanmaka olup, bölümlemeyi ve çoklamayı aynı anda gerçekleştirmektedir. İki aşamalı yöntemde ise, çoklama, K-bölümlü bir bölümlene elde edildikten sonra yapılmaktadır. Bu iki yöntemden, tek aşamalı yöntem PaToH hiperçizge bölümlene aracı içi tümleştirecek rpPaToH adında yeni bir çoklamalı bölümlene aracı genel kullanıma uygun olarak biçimde geliştirilmiştir.


7 REFERANSLAR


(GRAY, 1996) GRAY, Jim, Helland, Pat, Shasha, Dennis, “The Dangers of Replication And A Solution”, 1996 ACM SIGMOD International Conference on Management of Data


(SAITO, 2005) SAITO, Yasushi, Shapiro, Marc, “Optimistic Replication”, ACM Computing Surveys (CSUR), March 2005


(ROWSTRON, 2001c) ROWSTRON, A. and Druschel, P., "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, October 2001


Engineering @ Facebook’s Notes, Chat Stability and Scalability, http://www.facebook.com/note.php?note_id=51412338919


<table>
<thead>
<tr>
<th><strong>Proje No:</strong></th>
<th>109E019</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Proje Başlığı:</strong></td>
<td>Bellek Sınırılması Altında Etkin Paralel Veri Erişimi İçin Seçici Veri Replikasyonu</td>
</tr>
<tr>
<td><strong>Proje Yürütücüsü ve Araştırmacılar:</strong></td>
<td>Prof. Dr. Cevdet Aykanat, Ata Türk, Reha Oğuz Selvitopi, Volkan Yazıcı</td>
</tr>
<tr>
<td><strong>Projenin Yürütüldüğü Kuruluş ve Adresi:</strong></td>
<td>Bilkent Üniversitesi (Üniversiteler Mh., 06800 Ankara, Türkiye)</td>
</tr>
<tr>
<td><strong>Destekleyen Kuruluş(ların) Adı ve Adresi:</strong></td>
<td>TÜBİTAK (Remzi Oğuz Anı Mh., Tunus Caddesi 80, Ankara, Türkiye)</td>
</tr>
<tr>
<td><strong>Projenin Başlangıç ve Bitiş Tarihleri:</strong></td>
<td>01/09/2009 - 01/09/2011</td>
</tr>
</tbody>
</table>

Öz (en çok 70 kelime)

Projede terim-tabanlı bölümlenmiş paralel sorgu işleme sistemlerinde hiperçizge modellerini baz alarak akıllı ve seçici çoklama yapabiliyebilen teknikler geliştirilmiş ve gerçek zamanlı paralel sorgu işleme modellerinde test edilerek geçerlilikleri kanıtlanmıştır. Verilen bir hiperçizgenin düğümlerini çoklayan tek-aşamalı ve iki-aşamalı yöntemler öne sürülmüş ve tek-aşamalı yöntemin genel kullanıma uygun rpPaToH adında bir araç geliştirilmiştir. Deneylerle önerilen metodların şu anda paralel sorgu işleme sistemlerinde kullanılan modern çoklama metodlarından daha iyi olduğu gösterilmiştir.

**Anahtar Kelimeler:**
Paralel Sorgu İşleme, Hiperçizge, Çoklama, Sezgisel

**Fikri Ürün Bildirim Formu Sunuldu mu?** Evet □ Gerekli Değil □

Fikri Ürün Bildirim Formu'nun tesliminden sonra 3 ay içerisinde patent başvurusu yapılmalıdır.

**Projeden Yapılan Yayınlar:**

“Replicated Partitioning for Undirected Hypergraphs” (Reha Oğuz Selvitopi, Ata Türk, Cevdet Aykanat). Journal of Parallel and Distributed Computing (JPDC) dergisine sunulmuş ve “major revision” ile kabul edilmiştir.

“A Matrix Partitioning Interface to PaToH in MATLAB”, Parallel Computing Journal (2011)

“Query forwarding in geographically distributed search engines”, ACM SIGIR (2010)


“Parallel Frequent Item Set Mining with Selective Item Replication”, IEEE Transactions on Parallel and Distributed Systems (2011)


“Replicated Hypergraph Partitioning”, MSc. Thesis, Reha Oğuz Selvitopi


Ekte Bulunan “ARDEB Başarı Öyküsü Formu”, “Kazanımlar” Bölümünde Belirtilen Kriterlere Göre Proje Çıktılarınızın Başarı Öyküsü Niteliği Taşdığıni Düşünüyoruz

“ARDEB Başarı Öyküsü Formu”nu doldurunuz.
A Parallel Framework for In-Memory Construction of Term-Partitioned Inverted Indexes

TAYFUN KUCUKYILMAZ, ATA TURK AND CEVDET AYKANAT*

Computer Engineering Department, Bilkent University, 06800 Ankara, Turkey
*Corresponding author: aykanat@cs.bilkent.edu.tr

With the advances in cloud computing and huge RAMs provided by 64-bit architectures, it is possible to tackle large problems using memory-based solutions. Construction of term-based, partitioned, parallel inverted indexes is a communication intensive task and suitable for memory-based modeling. In this paper, we provide an efficient parallel framework for in-memory construction of term-based partitioned, inverted indexes. We show that, by utilizing an efficient bucketing scheme, we can eliminate the need for the generation of a global vocabulary. We propose and investigate assignment schemes that can reduce the communication overheads while minimizing the storage and final query processing imbalance. We also present a study on how communication among processors should be carried out with limited communication memory in order to reduce the total inversion time. We present several different communication-memory organizations and discuss their advantages and shortcomings. The conducted experiments indicate promising results.

Keywords: index inversion; term-based partitioning; parallel inversion; memory-based inversion

Received 30 July 2011; revised 29 October 2011
Handling editor: Yannis Manolopoulos

1. INTRODUCTION

The evolution of communication technologies in recent years gave rise to a rapid increase in the amount of textual digital information and the demand to search over this type of information. One of the largest industries of our era, the searching industry, has flourished around these demands.

Inverted indexes, due to their superior performance in answering phrase queries [1], are the most commonly used data structures in Web search systems. An inverted index consists of two parts: a vocabulary and inverted lists. The vocabulary contains the collection of distinct terms, which are composed of character strings (words) that occur in the documents of the collection. For each term in the vocabulary, there is an associated inverted list, or posting list. The inverted list for a term is a list of postings, where a posting contains an identifier for a document that contains that term. Depending on the granularity of information, the frequency and the exact term positions may also be stored in the postings.

Inverted index data structure is quite simple, yet Web-scale generation of a global inverted index is very costly due to the size, distributed nature and growth/change rate of the Web data [2]. Fast and efficient index construction schemes are required to provide fresh and up-to-date information to users. Furthermore, since the data to be indexed is crawled and stored by distributed or parallel systems (due to performance and scalability reasons), parallel index construction techniques are essential.

There are two major partitioning schemes used in distributing the inverted index on parallel systems: document-based and term-based partitioning. In document-based partitioning, the documents are assigned to index servers and all the postings related with the assigned documents are stored in a particular index server. In term-based partitioning, each term in the vocabulary and the related inverted lists are assigned to an index server.

Almost all of the major search engines use document-based partitioning due to the ease in parallel index construction of document-based, partitioned, inverted indexes. Term-based partitioning, on the other hand, has advantages that can be exploited for better query processing [3]. In this study, we propose an efficient parallel index construction framework that can be used for generating term-based partitioned, inverted
indexes starting from a document-based partitioned collection most possibly generated via a parallel crawling of Web documents.

1.1. Related work

Early studies on index construction are focused around disk-based algorithms designed for sequential systems [4–6]. In [4], authors present a method that traverses the disk-based document collection twice; once for generating a term-based partition to divide the work into loads, and once for inverting the dataset iteratively for each pass defined in the previous pass. The emphasis is on using as little memory as possible. In [5], authors use a multi-way, in-place, external merge algorithm for inverted index construction with less primary memory. In [6], authors propose an in-memory index construction method for disk-based inverted indexes where the document set is divided into batches that are inverted in memory and then merged and written into disk. In their work, authors facilitate the use of compression in order to achieve a more effective inversion.

More recent works on sequential systems are mainly focused on on-line incremental updates over disk-based, inverted indexes [7, 8]. In [7], the authors propose a hybrid indexing technique. The proposed method merges small posting lists with the already existing index, while using posting list reallocation for large posting lists. The authors also propose two in-place merge techniques for updating long posting lists. In [8], the authors evaluate two index maintenance strategies and propose alternatives for improving these strategies. These improvements are based on over-allocation of posting lists and keeping incremental updates within vocabulary before index remerging.

The following studies on index construction [3, 9–14] extend disk-based techniques for parallel systems. In [9], a document-based allocation scheme for inverted indexes is presented. The authors emphasize both storage balance and inter-processor communication times and try to minimize both using genetic algorithms. In [10], the authors evaluate the effects of term- and document-based partitioning methods on a shared-everything architecture. They use query statistics to balance the required I/O times among processors on a disk-based architecture.

In [11, 12], the authors present a disk-based, parallel index construction algorithm, where initially the local document collections are inverted by all processors in parallel. The processors generate a global vocabulary on a host processor and the host processor divides the document collection among all processors in lexicographic order assuming global knowledge over the document collection. The authors also analyze the merging phase of the inverted lists in [12], presenting three algorithms. In their work, the authors mainly focus on the parallel generation of the distributed index and the communication costs are not taken into consideration.

In [13], the author describes an index inversion framework for distributed information retrieval systems. Although the method presented in [13] achieves storage balance among processors, it does not consider minimizing the communication loads of the processors. In [13], it is also assumed that it is possible for the inverted indexes to be incrementally updated over time, and specialized data structures for minimizing the index update times are proposed. The cost of the inversion process is also emphasized, and four different index inversion methods are presented. In [14], the authors again start from a document partitioned collection and use a software-pipelined architecture to invert document collections. The collection is divided into runs, and for each run, documents are parsed, inverted, sorted and flushed into disk in a pipelined fashion. In [3], the authors propose a load balancing strategy in a term-partitioned inverted index on a pipelined query processing architecture [15]. In [3], both replication of inverted lists and a query statistics-based assignment scheme is presented, yielding up to 30% net query throughput improvement.

1.2. Motivation and contributions

We would like to repeat a catchy phrase often credited to Jim Gray: ‘Memory is the new disk, disk is the new tape’. With the advent of 64-bit architectures, huge memory spaces are available to single machines and even very large inverted indexes can fit into the total distributed memory of a cluster of such systems, enabling memory-based index construction. Furthermore, cloud computing systems such as Amazon EC2 are commercially available today. They offer leasing of virtual machines without owning and maintenance costs and thus ease the utilization and management of large cluster of servers. Thus, we believe that the benefits of parallel index construction is not limited to dividing and distributing the computational task to different processors. The current advances in network technologies, cloud computing and the high availability of low-cost memory provides an excellent medium for memory resident solutions for parallel index construction.

In this work, we extend our previously proposed in-memory parallel inverted index construction scheme [16] and compare the effects of different communication-memory organization schemes to the parallel inversion time. In our framework, we propose to avoid the communication costs associated with global vocabulary construction with a term-to-bucket assignment schema. This schema prevents term information to be sent to a host, where a reasonable term-to-processor assignment would be computed using the term distribution among processors, thus avoiding a possible bottleneck of communication. Furthermore, term-to-bucket partitioning allows the framework to completely avoid creating a global vocabulary, eliminating the need of a further communication phase.

We also investigate several assignment heuristics for improving the final storage balance, the final query processing loads and the communication costs of inverted index construction. Here, storage balance is important since we
are trying to build a memory-based inverted index. Query processing load balance is important since the reason for building the inverted index is for faster query processing and this can be done better if the loads of the processors are balanced. Finally, the communication cost is important since it affects the running time of parallel inversion.

Furthermore, we investigate the effects of various communication-memory organization schemes. Since parallel inversion is a communication-bound process, we observe that the utilization of the communication memory and the network has significant effects on the overall inversion time. Our findings indicate that, dividing the communication memory into \(2 \times K\) buffers, where \(K\) of which are used for sending messages and the remaining \(K\) are used for receiving messages, yields the best performance. This is due to the fact that this communication-memory organization scheme maximizes the communication/computation overlap.

Finally, we test the performance of the proposed schemes by performing both simulations and actual parallel inversion of a realistic Web dataset and report our observations. Our contributions in this work are prior to optimizations such as compression [17]. However, it is possible to apply data compression to the proposed model, making it possible to work with even larger data collections.

The organization of this paper is as follows: in Section 2, we introduce the memory-resident distributed index inversion problem and describe our framework. In Section 3, we provide our overall parallel inversion scheme. In Section 4, we describe the investigated assignment schemes in detail. In Section 5, we present several memory organization schemes in order to reduce the communication time and discuss their advantages and disadvantages. We provide our experiments, their analysis and extensive discussions on the results of our experiments in Section 6. Finally, in Section 7, we conclude and discuss some future work.

2. FRAMEWORK

Most of the largest text document collections that are actively in use today are Web-based. These repositories are mainly created and used by Web search engines. An important consideration in the design of parallel index construction systems should be their applicability to such real-life data collections. In this work, our efforts are based on presenting an efficient and scalable index construction framework specifically designed for Web-based document collections.

Parallel search engines collect Web pages to be indexed via distributed Web Crawlers [18]. In general, at the end of a crawling session, a document-based partition of the whole document collection is obtained, where each part is stored in a physically separate repository [18]. The state-of-the-art approach to distributing the crawling and storage tasks uses a site-hash-based assignment, where the site names of pages are hashed and documents are assigned to repositories according to those hash values [19–21].

The framework presented in this study has three assumptions on the initial data distribution. First, the initial document collection is assumed to be distributed among the processors of a parallel system; that is, each processor is assumed to have a portion of the crawled Web documents and maintain information about only its own local dataset. Thus, in this work, no processor contains a global view of the document collection. Secondly, each processor is assumed to contain a disjoint set of documents. This means that the overall system contains no replica of any document. Thirdly, the Web pages are assumed to be distributed among these processors using a site-based hashing; that is, all pages from a site are assigned to a single processor, and hence each site is assumed to be an atomic storage task. Consequently, the initial storage loads of the processors are not necessarily perfectly balanced. These three assumptions are in concordance with the output format of general purpose crawling systems.

In this framework, the objective of parallel index construction is to generate a final term-partitioned, parallel inverted index from a document-partitioned collection stored on a distributed shared-nothing architecture. The final term-partitioned inverted index will also be stored in a distributed fashion in order to allow both inter- and intra-query parallelism on query processing. In this context, our approach has similarities with parallel matrix transpose operations.

3. PARALLEL INVERSION

Our inversion scheme starts with a document-based, initial partition. Such an initial, document-based partition is depicted in Fig. 1a. Our overall parallel inversion scheme has the following phases:

(i) **Local inverted index construction**: Each processor generates a local inverted index from its local document collection. This process is illustrated in Fig. 1b. Note that inverted lists for some terms can appear in multiple processors.

(ii) **TermBucket-to-processor assignment**: Each processor uses hashing to find a deterministic assignment of terms into a predetermined number buckets. Buckets are used to randomly group inverted lists so that the communication costs in the termBucket-to-processor assignment phase is reduced. All processors communicate the sizes of their term buckets to the host processor. The host processor generates a termBucket-to-processor mapping under the constraint that in the final assignment, the storage and query processing load balance is achieved and communication cost is minimized. This process is illustrated in Fig. 1c. Note that many buckets exist in multiple processors due to the initial document partitioning.
(iii) Inverted list exchange-and-merge: The processors communicate appropriate parts of their local inverted indexes in an all-to-all fashion. This process is illustrated in Fig. 1d. The remaining local inverted index portions are merged with the received portions and final inverted index is generated. The final term-partitioned inverted index of the initial document-partitioned index in Fig. 1b can be seen in Fig. 1e.
3.1. Local inverted index construction

In the local inverted index construction phase, each processor generates a local vocabulary and local inverted lists from its local document collection. Since each processor only contains a unique subset of documents, this operation can be achieved concurrently without any communication. In this phase, the local vocabularies and inverted list sizes are determined and each term is given a unique identifier.

3.2. TermBucket-to-processor assignment

After the local inversion phase, processors contain a document-based, partitioned inverted index. In this partition, processors contain different portions of inverted lists for each term. In order to create a term-based, partitioned inverted index, each inverted list, in its full form, should be accumulated in one of the processors. To this end, each term in the global vocabulary should be assigned to a particular processor.

This term-to-processor assignment depicts an inverted index partitioning problem. A suitable index partitioning can be defined by many different criteria. In this work, we set the following quality metrics for a ‘good’ term-to-processor assignment:

QM1: Balancing the ‘expected’ query processing loads of processors.
QM2: Balancing the storage loads of processors.
QM3: Reducing the communication overhead during the inversion process through minimizing:

(a) Total communication volume.
(b) Communication load of the maximally loaded processor.

The final query processing loads of processors indicate the amount of processing that a processor is expected to perform once the inversion is finished and the query processing begins. We can estimate this load utilizing previous query logs.

The storage balance of processors guarantees an even distribution of the final inverted index allowing larger indexes to fit in the same set of processors.

Since inversion is a communication-bound process, the minimization of the communication overhead ensures that the inverted list exchange phase of the parallel inversion process takes less time. In this work, minimization of the communication overhead is modeled as the minimization of the total communication volume while maintaining the balance on the communication loads of the processors. These are the two commonly used quality metrics that determine the communication performance of a task-to-processor assignment when the message latency overhead remains negligible compared with the message volume overhead [22, 23], which is the case for parallel index inversion.

To optimize the above-mentioned metrics, we investigate existing assignment schemes, comment on possible enhancements over these schemes and propose a novel assignment scheme that performs better than its counterparts. Our discussions about bucket-to-processor assignment schemes are explained in detail in Section 4.

For the purpose of finding a suitable term-to-processor assignment, the previous works in the literature either assume the existence of a global vocabulary or generate a global vocabulary from the local vocabularies. The global vocabulary can be created by sending each term string, in its word form, to a host processor, where they are assigned global term-ids, and these global term-ids are broadcasted to all processors. However, in such a scheme, a particular term would be sent to the host machine by all processors if all processors contain that specific term. Our observations indicate that the cost of such an expensive communication stage is proportional to the cost of inverted list exchange phase. Furthermore, since the host processor receives all the communication, it constitutes a serious bottleneck.

In this work, we propose a novel and intelligent scheme that enables us to avoid global vocabulary construction cost. We propose to group terms into buckets prior to the term-to-processor assignment. Using string hashing functions, each word in a local vocabulary is assigned to a bucket. Afterward, only the bucket size information is sent to the host processor. The host processor computes a termBucket-to-processor assignment, which induces a term-to-processor assignment, and broadcasts this information to the processors. The effect of the bucket processing order on the quality of the assignment is not investigated in this work and the same random bucket processing order is used in evaluating the assignment schemes. We should also note here that it is not necessary to build a global index at the host processor ever. It suffices for the host processor to store only a bucket-to-processor assignment array. Whenever the host processor receives a query term, all it has to do is to compute the hash of the term, find the bucket for that term and forward the term to the owner processor of the bucket.

3.3. Inverted list exchange-and-merge

At the end of the termBucket-to-processor assignment phase, all bucket-to-processor assignments are broadcast to the processors by the host processor, so that each processor is aware of the bucket-to-processor assignments. In order to create a term-partitioned inverted index, the document-based partitioned, local inverted list portions should be communicated between processors in such a way that the whole posting list of each term resides in one of the processors. To this end, all processors should exchange their inverted list portions in an all-to-all fashion. However, utilizing termBuckets instead of terms for assignment dictates a major change (and an additional cost) in the inverted list exchange-and-merge phase.
Since termBucket-to-Processor assignment prevents the need of creating a global vocabulary, when a processor receives a posting list portion of a term from another processor, it also requires additional information to identify the posting list it receives. To this end, upon sending the posting list portions, the processors should also send the associated term, in its word form, to the receiving processor. Due to this, the all-to-all inverted list exchange communication becomes slightly more costly. However, since the processor-to-host bottleneck due to global vocabulary construction is already avoided, the performance degradation in all-to-all inverted list exchange communication is more than compensated. Furthermore, this vocabulary exchange is distributed among all processors evenly, further reducing its overhead.

The inverted list exchange between processors is achieved in two steps. First, terms, in their word form, and their posting sizes are communicated. This is done by an all-to-all personalized communication phase, where each processor receives a single message from each other processor. At the end of this step, all processors obtain their final local vocabularies and can reserve space for their final local inverted index structures. Secondly, inverted list portions are exchanged in bucket id order, and within the buckets in alphabetical order. This step is again performed as an all-to-all personalized communication. However, since this step consumes significant amount of time, the inverted list portions are sent via multiple messages. Memory organization and communication scheme used in this phase is explained in detail in Section 5. At the end of inverted list exchange, the remaining inverted lists and obtained inverted lists for each term are merged and written into their reserved spaces in the memory.

4. TERM-TO-PROCESSOR ASSIGNMENT SCHEMES

In this section, we try to solve the termBucket-to-processor assignment problem with the objectives of minimizing the communication overhead during the inversion and maintaining a balance on the query processing and storage loads of processors after the inversion. We present adaptations of two previously proposed assignment algorithms [24] to the problem at hand, discuss the shortcomings of these algorithms and propose a novel assignment algorithm that provides superior parallel performance.

In the forthcoming discussions we use the following notations: The vocabulary of terms is indicated by \( \mathcal{T} \). Due to the initial site-hash-based crawling assumption, the posting list of each term \( t_j \in \mathcal{T} \) is distributed among the \( K \) processors. In this distribution, \( w_k(t_j) \) denotes the size of the posting list portion of term \( t_j \) that resides in processor \( p_k \) at the beginning of the inversion, whereas \( w_{tot}(t_j) = \sum_{k=1}^{K} w_k(t_j) \) denotes the total posting list size of term \( t_j \).

We assume that prior to bucket-to-processor assignment, each processor has built its local inverted index \( \mathcal{I}_k \) and partitioned the vocabulary \( \mathcal{T} = \{t_1, t_2, \ldots, t_n\} \) containing \( n \) terms into a predetermined number \( m \) of buckets. The number of buckets \( m \) is selected such that \( m \ll n \) and \( m \gg K \). Let

\[
\mathcal{B} = \Pi(\mathcal{T}) = \{\mathcal{T}_1 = b_1, \mathcal{T}_2 = b_2, \ldots, \mathcal{T}_m = b_m\}. \tag{1}
\]

denote a random term (RT)-to-bucket partition, where \( \mathcal{T}_i \) denotes the set of terms that are assigned to bucket \( b_i \). In this partition, \( w_{tot}(b_i) \) denotes the total size of the posting lists of terms that belong to \( b_i \) and \( w_k(b_i) \) denotes the total size of the posting list portions of terms that belong to \( b_i \) and that reside in processor \( p_k \) at the beginning of the inversion.

We also assume that we are given a query set \( Q \) where each query \( q \in Q \) is a subset of \( \mathcal{T} \), i.e. \( q \subseteq \mathcal{T} \). The number of queries that a term \( t_j \) is requested by is denoted with \( f(t_j) \).

In an \( m \)-bucket and \( K \)-processor system, the bucket-to-processor assignment can be represented via a \( K \)-way partition of the buckets among the processors. The quality of a bucket-to-processor assignment \( \Pi(B) \) is measured in terms of three metrics: The query processing load balance \( QM_1 \), storage load balance \( QM_2 \) and the communication cost \( QM_3 \). The query processing load \( \mathcal{Q}(p_k) \) of a processor \( p_k \) induced by the assignment \( \Pi(B) \) is defined as follows:

\[
\mathcal{Q}(p_k) = \sum_{b_i \in \mathcal{B}_k} \sum_{t_j \in b_i} w_{tot}(t_j) \times f(t_j). \tag{3}
\]

The storage load \( S(p_k) \) of a processor \( p_k \) induced by the assignment \( \Pi(B) \) is defined as follows:

\[
S(p_k) = \sum_{b_i \in \mathcal{B}_k} \sum_{t_j \in b_i} w_{tot}(t_j). \tag{4}
\]

The communication cost of a processor \( p_k \) induced by the assignment \( \Pi(B) \) has two components. Each processor must receive all portions of the buckets assigned to it from other processors. Thus total reception cost/volume of a processor \( p_k \) is

\[
\text{Recv}(p_k) = \sum_{b_i \in \mathcal{B}_k} \sum_{t_j \in b_i} (w_{tot}(t_j) - w_k(t_j)). \tag{5}
\]

Each processor must also send all postings that are not assigned to it to some other processor. The total transmission cost of \( p_k \) is represented by \( \text{Send}(p_k) \) and is defined as

\[
\text{Send}(p_k) = \sum_{b_i \notin \mathcal{B}_k} \sum_{t_j \in b_i} w_k(t_j). \tag{6}
\]

The total communication cost of a processor is defined as

\[
\text{Comm}(p_k) = \text{Send}(p_k) + \text{Recv}(p_k). \tag{7}
\]
4.1. Minimum communication assignment

Minimum communication assignment (MCA) algorithm minimizes the total communication volume while ignoring storage and communication balancing [24]. The MCA scheme is based on the following simple observation. If a termBucket is assigned to the processor that contains the largest portion of the inverted lists of the terms belonging to that bucket, the total message volume incurred due to this assignment will be minimized. Thus, if we assign each termBucket \( b_i \in B \) to the processor \( p_k \) that has the largest \( \omega_i(b_i) \) value, the total volume of communication for this term will be minimized. By assigning all terms using the above criteria, an assignment with global minimum total communication volume can be achieved.

4.2. Balanced-load MCA

The balanced-load (BLMCA) scheme is an effort to incorporate storage balancing to MCA [24]. In this scheme, termBuckets are iteratively assigned to processors. In BLMCA, for each termBucket, first the target processor that will incur the minimal total communication is determined using the criteria in the MCA scheme. If assignment of the particular termBucket to that processor does not make the storage loads of the processors more skewed (does not increase the maximum storage load of all processors) at that iteration, the assignment proceeds as in the MCA scheme. Otherwise, the termBucket is assigned to the minimally loaded processor.

4.3. Energy-based assignment

In BLMCA, two separate cost metrics are evaluated: The storage load balance and total communication cost. However, at each iteration, only one of these metrics is chosen to be optimized. Furthermore, both MCA and BLMCA model the communication cost as the total communication volume and disregard the maximum communication volume of a single processor. In order to minimize the maximum communication cost of a processor, we should consider both the reception cost of the assigned processor and the transmission costs of all other processors.

In the energy-based assignment (EA) scheme, we propose a model that prioritizes reducing the maximum communication cost of processors as well as maintaining storage and query processing load balance. To this end we define the energy \( E \) of an assignment \( \Pi(B) \). This energy definition is based on the storage loads, query processing loads and communication costs of processors. Recall that \( \text{Comm}(p_k) \) of a processor incorporates both reception and transmission costs of processor \( p_k \). We define two different energy functions for a given termBucket-to-processor assignment \( \Pi(B) \):

\[
E^1(\Pi(B)) = \max \left\{ \max_{1 \leq k \leq K} \{\text{Comm}(p_k)\}, \max_{1 \leq k \leq K} \{\text{QP}(p_k)\} \right\},
\]

\[
E^2(\Pi(B)) = \sum_{k=1}^{K} (\text{Comm}(p_k))^2 + \sum_{k=1}^{K} (S(p_k))^2 + \sum_{k=1}^{K} (\text{QP}(p_k))^2.
\]

Utilizing these two energy functions, we propose a constructive algorithm that assigns termBuckets to processors in a successive fashion. The termBuckets are processed in some order, and the energy increase in the system by \( K \) possible assignments of each bucket are considered. The assignment that incurs the minimum energy increase is performed; that is, for the assignment of a termBucket \( b_i \) in the given order, we select the assignment that minimizes

\[
E(\Pi(B_{i-1} \cup \{b_i\})) - E(\Pi(B_{i-1})),
\]

where \( B_{i-1} \) denotes the set of already assigned termBuckets.

We should note here that proposed EA schemes also have the nice property of being easily adaptable for incremental index updates. To enable this feature at the end of inversion process, it is sufficient to store the energy levels of each process. These values then can be used to perform (re)assignment of indexes in an incremental fashion. The minimization of inversion time feature of these schemes would be very helpful in minimizing the incremental update time as well. However, we should note that enabling incremental update in these schemes would necessitate the construction of a global vocabulary on the server node.

We consider both \( E^1 \) and \( E^2 \) energy definitions and report the results of both schemes in our experiments. We call the \( E^1 \)-based assignment scheme as \( E^1A \) and the \( E^2 \)-based assignment scheme as \( E^2A \).

5. COMMUNICATION-MEMORY ORGANIZATION

In the final stage of the memory-based parallel inverted index construction, the portions of each posting list are communicated between processors to accumulate each posting list in one processor, where they would be merged in order to construct the final inverted index. This phase can be summarized as an all-to-all personalized communication phase with different number of messages and total message sizes. In this phase, each processor should identify local posting list portions to be sent to other processors, prepare message buffers to send them using the available memory for this communication and send them to the target processors. At the same time, each processor should retrieve posting list portions assigned to them from other processors and merge them in order to generate the final posting lists.

Posting list exchange operation requires intensive communication between processors and dominates the total time required to complete the index inversion. An important question when communicating the posting list portions is how to use/organize
the available memory so that the communication phase takes the least possible time. In this work, we evaluate four different communication memory organization schemes and their impact on the total run time of index inversion. These schemes are:

(i) 1-Send 1-Receive buffer scheme (1s1r).
(ii) 1-Send \((K - 1)\)-Receive buffer scheme (1sKr).
(iii) \((K - 1)\)-Send 1-Receive buffer scheme (Ks1r).
(iv) \((K - 1)\)-Send \((K - 1)\)-Receive buffer scheme (KsKr).

In investigating different communication-memory organization schemes, we assume that the total memory spared for communication is fixed, say \(M\). In 1s1r, the communication memory is split into one send and one receive buffer, each with size \(M/2\). In 1sKr and Ks1r, the memory is split into \(K\) buffers each with size \(M/K\). In 1sKr, one of these buffers is used as a send buffer and the remaining \(K - 1\) buffers are reserved for receiving messages from other processors. In Ks1r, each processor maintains one receive buffer and \(K - 1\) send buffers, which are reserved for sending messages to other processors. In KsKr, the memory is split into \(2 \times K\) buffers each with size \(M/((2 \times K) - 2)\). \(K - 1\) of these buffers are reserved as send buffers as in Ks1r, while the other \(K - 1\) buffers are reserved as receive buffers as in 1sKr.

In all of these schemes, the communication commences through several stages. First, all processors issue non-blocking receives for each receive buffer. Then, each processor starts preparing the outgoing send buffer(s). During this preparation, the vocabulary of the local inverted index is traversed in order to copy the local posting list portions to the send buffer(s). Whenever a send buffer is full, the owner processor issues a blocking send operation. The blocking send operation stalls all computation on the sender-side until the send operation is successfully completed. Upon receiving a message, each processor starts emptying its respective receive buffer by copying the received posting list portions to the final inverted index, effectively finalizing the merge of posting list portions. After the merging phase is completed, processors issue a new non-blocking receive in order to receive any remaining messages from other processors, and restart filling their send buffers.

5.1. 1-Send (1s) versus \((K - 1)\)-send (Ks) buffer schemes

In the 1s buffer schemes, in order to prepare messages to be sent to other processors, all posting list portions targeted to a specific processor should be put into the single send buffer prior to sending it. For a single target processor, in order to send all required posting list portions, the vocabularies of each local inverted index must be traversed once. As each processor probably requires to communicate with all other processors, preparation of the send buffers requires \(K - 1\) traversals over the local inverted index.

On the other hand, in the Ks buffer schemes, in order to prepare outgoing messages, only one traversal of the local inverted index is sufficient. In this traversal, the processor would pick any outgoing posting list portion and place it into the appropriate send buffer. Once one of the send buffers is full, the communication can commence. However, using blocking sends ultimately results in stalling the process every time a send is issued, reducing the processor utilization.

5.2. 1-Receive (1r) versus \((K - 1)\)-receive (Kr) buffer schemes

In 1r schemes, the communication memory is fairly utilized, whereas in Kr schemes, the utilization of the communication memory depends on the number of messages received by each processor and may be poor for most of the processors. In 1sKr, since there can be only \(K\) messages over the network at any time, only \(K\) of the \(K \times (K - 1)\) receive buffers would be actively used. In this case, \(K \times (K - 2)\) unused receive buffers are left idle, leaving the \((K - 2) \times M\) of the total \(K \times M\) memory unused. In KsKr, since there are \(K - 1\) send buffers, the processors can produce enough messages to actively use most of the \(K \times (K - 1)\) receive buffers, resulting in a more utilized communication memory.

In Kr schemes, since each processor has a specific receive buffer for all other processors, cycles in the communication dependency graph do not cause deadlocks. However, in 1r schemes, depending on the communication order, cycles in the communication dependency graph may cause deadlocks. To avoid these deadlocks, we can utilize non-blocking sends instead of blocking sends. Non-blocking sends allow a processor to continue processing after a send is issued without the need of waiting for it to finalize, thus avoiding any possible deadlocks. However, the issued send still requires its particular send buffer to be intact. As a result, the processor should again be halted in case a local posting list is required to be written in that send buffer. For this reason, each send buffer is locked after a send, and all such buffers are probed after each messaging iteration. If a send buffer is released after a successful send, the lock is freed, allowing the processor to issue writes into that send buffer again.

In Ks1r, whenever a non-blocking send is issued, it is possible to fill other send buffers, allowing computation to overlap with communication. However, in 1s1r, deadlock avoidance via non-blocking sends may cause poor performance since there is only one send buffer and it is not possible to overwrite the contents of this buffer until the non-blocking send is completed, causing the computation to be stalled.

It is also possible to avoid deadlocks in the 1s1r scheme by employing a BSP-like [25] communication/computation pattern and by ensuring that no two processors send messages to the same processor in any given communication step. In 1s1r, since \(K - 1\) traversals over the local inverted index is required for each processor, it is possible to divide the computation into \(K - 1\) traversal steps and communicate at the end of each computation step. We can also freely choose the communication...
A PARALLEL FRAMEWORK FOR IN-MEMORY CONSTRUCTION

6. EXPERIMENTS

6.1. Experimental framework

We conducted our experiments on a realistic dataset obtained by crawling educational sites across America. The raw size of the dataset is 30 GB and contains 1 883 037 pages from 18 997 different sites. The biggest site contains 10 352 pages while average number of pages per site is 99.1. The vocabulary of the dataset consists of 3 325 075 distinct terms. There are 787 221 668 words in the dataset. The size of the inverted index generated from the dataset is 2.8 GB. For query load balancing purposes, we used a synthetically generated query log of 1 000 000 distinct queries each of which contains 1 to 7 terms. In our experiments, we used a fixed number of buckets in termBucket-to-processor assignment and set the number of buckets to 10 000.

We tested the performance of the proposed assignment schemes in two different ways: First we report the relative performances of the assignment schemes in terms of the quality metrics described in Section 3.2 through simulations. In simulations we theoretically compute the assignment of terms to processors and compute the storage, query processing and communication costs of the assignment without performing actual parallel inversion. The simulation experiments are conducted for \( K = \{4, 8, 16, 32, 64, 128\} \) values on a Sun AMD-Opteron machine with 128 GB of RAM.

Secondly, we provide a set of experiments using actual parallel inversion runs in order to show how improvements in quality metrics relate to parallel running times. For this purpose we developed an MPI-based parallel inversion code that can utilize each of the four communication-memory organization schemes described in Section 5 for a given termBucket-to-processor assignment. These second set of experiments are conducted on a 32-node PC-cluster, where each node is an Intel Pentium IV 3.0 GHz processor with 1 GB RAM connected via an interconnection network of 100 Mb/s fast Ethernet. The total communication-memory size \( M \) is set at 5 MB in these experiments.

6.2. Evaluation of the assignment schemes

As a baseline inversion method, we implemented a RT assignment algorithm. In the RT scheme, each term is assigned to a random processor without a term-to-bucket assignment. In this scheme the global vocabulary has to be created. In order to evaluate the viability of term-to-bucket assignment and as a baseline termBucket-to-processor assignment scheme, we also implemented a random assignment (RA) algorithm that assigns buckets to processors randomly. Note that RA requires the least possible time to compute a termBucket-to-processor assignment while avoiding the need for global vocabulary creation, and thus it can be used to compare/analyze the merits of the proposed bucketing scheme and the assignment schemes. The performance of the proposed assignment schemes are compared against the RT and RA schemes.

6.2.1. Simulation results

Tables 1–3 compare the performance of the assignment schemes in terms of the quality metrics described in Section 3.2.

Table 1 displays the performance of the proposed assignment schemes in optimizing the quality metric \( QM_1 \). In the table, the query load imbalance percentages for different assignment schemes and different number of processors is presented. The
query load imbalance values are calculated according to the following formula:

\[
\frac{\left(\max_{1 \leq k \leq K} Q(p_k) - 1\right)}{\left(\sum_{k=1}^{K} Q(p_k)\right)/K} \times 100.
\]  

(11)

Table 2 shows the performance of the proposed assignment schemes in optimizing quality metric \( QM2 \). In the table, the initial imbalances due to hash-based distribution and the final imbalances after applying the assignment schemes are presented. The storage imbalance values are computed according to the following formula:

\[
\frac{\left(\max_{1 \leq k \leq K} S(p_k) - 1\right)}{\left(\sum_{k=1}^{K} S(p_k)\right)/K} \times 100.
\]  

(12)

Table 3 compares the communication performance of the assignment schemes in terms of the average and the maximum message volume to be handled by a processor during parallel index inversion. The total volume of communication required by an assignment scheme can be computed from the table by multiplying the respective average message volume value of the assignment scheme with the respective \( K \) value. Thus, the ‘Avg’ columns of Table 3 indicate the performance of the assignment schemes in optimizing \( QM3(a) \). The ‘Max’ columns in Table 3 indicate the communication load of the maximally loaded processor and thus indicate the performance of the assignment scheme in optimizing \( QM3(b) \). The communication-load balancing performance of each assignment scheme can be evaluated by comparing the ‘Avg’ and ‘Max’ columns.

The comparison of RT and RA schemes relates to the effectiveness of the proposed term-to-bucket assignment. As shown in Table 2, RA performs slightly better than RT for \( K \leq 64 \). Both Tables 1 and 3 display that RT and RA perform similarly in terms of query load balancing and communication volumes. Comparison of these two assignment schemes shows that term-to-bucket assignment prevents the global vocabulary construction without degrading much our quality metrics.

As seen in Table 1, MCA achieves significantly worse query load imbalance than all other assignment schemes. Similarly, Table 2 shows that MCA considerably degrades the initial storage balance. On the other hand, Table 3 reveals that MCA achieves the best average communication cost. These experimental findings are expected since MCA only considers the minimization of the total communication cost, disregarding storage and communication balancing.

As mentioned in Section 4.2, BLMCA is a modified version of MCA with added emphasis on storage balancing. As seen in Table 2, BLMCA achieves the best final storage balance in all instances. However, as seen in Table 3, the storage balance in BLMCA is achieved at the expense of increased...
total communication volume compared with MCA. Table 1 also shows that especially with increasing $K$, BLMCA fails in balancing query processing loads.

Table 1 displays that, for all processor values, $E^2 A$ performs significantly better than all other assignment schemes in terms of query processing load balance. Additionally, in terms of query load imbalances, $E^1 A$ is the second best performer. As seen in Tables 2 and 3, although $E^2 A$ slightly degrades the storage balance, it performs better than the other schemes in terms of maximum communication volume handled by a processor for almost all $K$ values (except for $K = 2$ and 4). Although $E^1 A$ produces better storage balance than $E^2 A$, the communication volume handled by a processor incurred by $E^1 A$ is slightly worse than BLMCA. In terms of the maximum communication volume handled by a processor, $E^2 A$ achieves the best results for $K > 8$. Table 3 also indicates that the average and maximum communication volume values induced by $E^2 A$ are close, which shows that $E^2 A$ manages to distribute the communication load among processors evenly.

6.2.2. Parallel inversion results

Table 4 compares the running times of our parallel inversion code for different assignment schemes. Since the creation of the local inverted indexes from local document sets is an operation prior to our inversion schemes, it is assumed that the local inverted indexes are already created. Thus, the time for converting local document collection to local inverted indexes is not included in the inversion times displayed in Table 4.

We provide the RT scheme in order to present the benefits of using a term-to-bucket assignment. The RT scheme differs from other schemes in two ways. First, in the RT scheme termBucket-to-Processor assignment is replaced with a term-to-processor assignment. Secondly, in the RT scheme there is an additional phase called global vocabulary construction phase. As seen in Table 4, RT performs significantly worse than other assignment schemes for all $K$ values other than $K = 2$. This indicates that our bucketing scheme has a significant impact on performance.

As seen in Table 4, for $K = 2$, MCA achieves the lowest inversion time compared with the other schemes. This is because, for $K = 2$, minimizing the total communication volume also minimizes the maximum communication volume handled by a processor. However, for all $K$ values greater than 2, MCA performs significantly worse since the maximum message volume handled by a processor for MCA is considerably higher than other assignment schemes. As seen in Table 4, $E^2 A$ performs considerably better than the other assignment schemes. For example for $K = 32$, $E^2 A$ performs up to 9% better than RA in terms of running time and achieves better final query load and storage balancing. The relative performance order of the assignment schemes in terms of actual inversion time values displayed in Table 4 are generally in concordance with the relative performance order of the assignment schemes in terms of the quality metrics displayed in Tables 2 and 3.

Figure 2 displays the dissection of parallel inversion time into: local inverted index construction, termBucket-to-Processor assignment and inverted list exchange-and-merge phases for different assignment and communication-memory organization schemes on $K = 8$ processors. For the sake of a better insight on the overall index inversion process, the inverted list exchange-and-merge phase is further divided into two components. The first component is called vocabulary communication, where processors send each other the terms, in their word form, and the associated posting list sizes in an all-to-all personalized fashion. The second component is called inverted list communication, where the posting list portions are communicated between processors.

Table 4 also shows that, as the complexity of the assignment schemes increases, the time required for termBucket-to-processor assignment also increases. The RA-based termBucket-to-processor assignment phase takes <1% of the total inversion time, whereas the $E^2 A$-based termBucket-to-processor assignment phase takes more than 4% of the total inversion time.

<table>
<thead>
<tr>
<th>$K$</th>
<th>RT</th>
<th>RA</th>
<th>MCA</th>
<th>BLMCA</th>
<th>$E^1 A$</th>
<th>$E^2 A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>105.90</td>
<td>109.80</td>
<td>85.72</td>
<td>106.08</td>
<td>108.15</td>
<td>108.13</td>
</tr>
<tr>
<td>4</td>
<td>71.60</td>
<td>69.19</td>
<td>81.34</td>
<td>68.63</td>
<td>69.34</td>
<td>68.49</td>
</tr>
<tr>
<td>8</td>
<td>68.44</td>
<td>51.42</td>
<td>66.76</td>
<td>46.45</td>
<td>47.27</td>
<td>45.74</td>
</tr>
</tbody>
</table>

TABLE 4. Parallel inversion times (in seconds) including assignment and inverted list exchange times for different assignment and communication-memory organization schemes.
FIGURE 2. Times (seconds) of various phases of the parallel inversion algorithm for different assignment and communication-memory organization schemes on $K = 8$ processors.

columns of Table 3 suggest, the time spent on the vocabulary communication phase is minimum for $E^2A$ and maximum for the MCA assignment scheme.

As seen in Fig. 2, the inverted list exchange-and-merge phase takes almost 85% of the total inversion time, thus confirming that parallel inversion is a communication-bound process. We compare and analyze the impact of different communication-memory organization schemes on this phase in the following subsection.

6.3. Evaluation of communication-memory organization schemes

Table 4 compares the running times of parallel inversion for different communication-memory organization schemes. $Ks1r$ has the worst overall performance for all $K$ values >2. Although $Ks1r$ avoids redundant memory reads by doing only one traversal over the local inverted lists, the use of blocking sends causes stalls and prevents overlap between communication and computation.

Although $1sKr$ performs better than $ls1r$ for $K \leq 16$, its relative performance decreases when the number of processors increases. This is due to lower memory utilization of $1sKr$ on a higher number of processors since each processor must maintain $K - 1$ receive buffers. We theorize that for a higher number of processors, $1sKr$ would perform even worse.

For all $K$ values >2, $KsKr$ performs superior with respect to the other communication-memory organization schemes. As the number of processors increases, the performance gap between $KsKr$ and the other schemes increases in favor of $KsKr$. This is because $KsKr$ avoids redundant traversals during the preparation of send buffers and overlaps computation with communication. For this reason, we select $KsKr$ as the default communication-memory organization scheme for the remaining experiment.

Figure 3 evaluates the effect of the available communication-memory size (M) on the running time of parallel inversion code utilizing the $E^2A$ assignment scheme and the $KsKr$ communication-memory organization scheme for $K = 8$ processors. As seen in Fig. 3, $KsKr$ scales well with increasing communication-memory size. The ability to continue to process several send buffers without stalling allows $KsKr$ to function relatively better with larger communication-memory sizes.

7. CONCLUSIONS

In this paper, a memory-based, term-partitioned, parallel inverted, index construction framework was examined. Several
problems were identified and improvements were proposed for a parallel index inversion framework.

First, we proposed a termBucket-to-processor assignment scheme. This scheme minimizes the communication cost of local vocabularies among processors and distributes the final query processing and storage loads among all processors, allowing a finer grained parallelism. We also showed that, by using a termBucket-to-processor assignment scheme, the need to create a global vocabulary can be eliminated and all associated communications can be prevented.

Secondly, we developed and investigated several heuristics for generating a term-to-processor assignment. The results of our experiments show that, compared with a baseline RA scheme, our proposed methods improved the parallel inversion times significantly while providing reasonable final query processing and storage balances.

Thirdly, we presented and explored four different communication-memory organization schemes in order to reduce the communication time required. We also presented methods to avoid deadlocks and network congestion and commented on memory utilization of the overall system. Our results show that splitting the communication-memory in 2 × (K − 1) parts yields the best results.

Simulations and actual parallel inversion times are presented in order to give insight on our improvements. According to the observed results, we recommend the use of the E^2A scheme for termBucket-to-processor assignment, and the K s K r scheme for communication-memory organization.

This work can be extended in several ways. First, the framework used in this work does not consider the effect of the bucket processing order. For example, processing buckets in decreasing size order might present better results both in respect of final storage balance and communication costs. Secondly, the number of buckets is assumed to be fixed throughout this work. The scaling of our framework using different numbers of buckets can also be considered.

FUNDING
This work is partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project EEEAG-109E019.

REFERENCES


Dear Author,

Please check your proof carefully and mark all corrections at the appropriate place in the proof (e.g., by using on-screen annotation in the PDF file) or compile them in a separate list. Note: if you opt to annotate the file with software other than Adobe Reader then please also highlight the appropriate place in the PDF file. To ensure fast publication of your paper please return your corrections within 48 hours.

For correction or revision of any artwork, please consult [http://www.elsevier.com/artworkinstructions](http://www.elsevier.com/artworkinstructions).

Any queries or remarks that have arisen during the processing of your manuscript are listed below and highlighted by flags in the proof. Click on the 'Q' link to go to the location in the proof.

<table>
<thead>
<tr>
<th>Location in article</th>
<th>Query / Remark: click on the Q link to go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Please confirm that given names and surnames have been identified correctly.</td>
</tr>
<tr>
<td>Q2</td>
<td>Please update reference ‘Blanco et al. (in press)’.</td>
</tr>
</tbody>
</table>

Thank you for your assistance.
Highlights

- Document replication strategies for geographically distributed search engines.
- Search quality, average query response time, and query workload criteria.
- Selective, partial document replication is superior to full or no replication.
- Experiments with a real-life setting and a large query log.
Document replication strategies for geographically distributed web search engines

Enver Kayaaslan a,1, B. Barla Cambazoglu b,*, Çevdet Aykanat a

a Computer Engineering Department, Bilkent University, Ankara, Turkey
b Yahoo! Research, Barcelona, Spain

1. Introduction

We consider a large-scale web search engine architecture with multiple, geographically distributed data centers (Baeza-Yates, Gionis, Junqueira, Plachouras, & Telloli, 2009; Cambazoglu, Plachouras, & Baeza-Yates, 2009). In this architecture, each data center crawls and maintains the documents that are served by the web sites in its geographical region (Cambazoglu, Plachouras, Junqueira, & Telloli, 2008). User queries are routed to data centers according to the regions they originate from. For example, a data center in Madrid crawls the web sites in Spain and processes the queries submitted from Spain. As we will discuss next, this architecture leads to two extremes for the placement of the web index and query processing.

At one extreme, a global index is built over the entire web collection, and this index is replicated on all data centers. Queries are processed on the entire web index, and hence search result qualities are identical to those of a centralized search architecture. However, this approach does not scale well since the global web index needs to be constructed from a distributed document collection and periodically maintained. Moreover, this approach requires major hardware investments and results in high power consumption, which is an important issue for commercial search engines. Finally, processing queries...
over an entire web index may be too costly to satisfy the tight response time constraints of large-scale web search engines (Cambazoglu, Zaragoza, et al., 2010).

At the other extreme, each data center builds a regional web index on its local crawl and processes its queries over this partial (local) index. This approach is highly scalable because partial indexes are locally maintained and less resources are needed for query processing (alternatively, queries can be processed faster). However, as processing of queries is limited to a partial index, some high-quality or best matching documents that are indexed by non-local data centers may be missing in search results. This may lead to not affordable losses in search result qualities, negatively impacting the user satisfaction and potentially the revenues of the search engine.

A search engine architecture based on selective replication of documents on data centers emerges as a feasible mid-ground between these two extremes. The main idea in selective replication is to identify the documents that are of interest to the users of each geographical region and replicate the documents on the data centers according to the user interest. If this can be wisely done, queries can be locally processed in regional data centers, reducing the search quality loss relative to the second extreme and providing better scalability compared to the first extreme. Selective replication can be further coupled with selective forwarding of queries between data centers so that documents that are missing in the local top k results (with respect to the global top k results) can be retrieved from non-local data centers, preventing any search quality loss (Baeza-Yates et al., 2009; Cambazoglu, Varol, Kayaaslan, Aykanat, & Baeza-Yates, 2010).

In this paper, we propose strategies for selectively replicating documents in a geographically distributed search engine setting. Our strategies identify the documents that are of interest to the users of certain geographical regions, based on the occurrence frequencies of documents in past search results. The identified documents are then replicated and indexed on non-local data centers so that future queries can be efficiently and effectively processed.

The outline of the paper is as follows. In Section 2, we provide the details of the search engine architecture that we consider in this work and provide formal definitions for two variants of the document replication problem we aim to solve. Section 3 describes the datasets used in our work and the setup of our simulations. In Sections 4–6, we propose various replication algorithms, each optimizing different performance metrics under different constraints and assumptions. We report the experimental results about the performance of our algorithms in the associated sections. In Section 7, we investigate the impact of query forwarding on the performance. We survey the related work in Section 8. The paper is concluded in Section 9.

2. Preliminaries

2.1. Architecture

We consider a search engine architecture composed of multiple data centers. In this architecture, each data center crawls and stores documents belonging to a disjoint subset of the Web. Each data center then builds a local web index over its crawled documents, independent of the other data centers. We assume that IP addresses (or countries at a higher granularity) are statically assigned to data centers according to their geographical proximity. Each data center is responsible for processing queries that originate from its subset of IPs and is said to be the local data center for those queries.

In our architecture, certain documents are replicated on non-local data centers. Hence, in addition to its local index, each data center maintains a replicated index, built over its non-local documents. The replication pattern of documents is periodically determined based on the frequencies with which documents appear in the search results generated by individual data centers.

Queries are evaluated as follows. A user query is first processed in the local data center associated with the user, over both local and replicated indexes. A local top k result set is formed based on the estimated relevance scores of documents (Cambazoglu & Aykanat, 2006). At this point, this result set may be immediately returned to the user. Alternatively, the query may be forwarded to a set of non-local data centers, hoping to retrieve some documents whose scores are higher than that of the lowest scoring document in the locally computed top k set. Forwarded queries are concurrently processed over the local indexes of non-local data centers, whose top k results are returned to the local data center. These results are then merged in decreasing order of scores and the top k results are returned to the user.

Fig. 1 illustrates the process. In the figure, each data center is represented by a large box. The patterns indicate the original assignment of documents to data centers. The box in the top row represents the local documents of a data center. The boxes in the bottom row represent non-local documents that are locally replicated. The directed arcs show contributions of different document collections to the final search results.

In this architecture, if a query is only locally processed, the search result quality may deteriorate as some of the documents that appear in the global top k result set may not be available in the local data center. On the other hand, if the query is forwarded to non-local data centers, the query response time increases due to the network latency between the local and non-local data centers (the workload may also increase). Readers may refer to (Cambazoglu, Varol, et al., 2010) for more details about the architecture.

We assume that the global collection statistics are made available to all data centers so that the scores generated by different data centers are comparable.
We note that document replication and query forwarding are not competing strategies, but rather query forwarding is a technique that is complementary to partial document replication. There are query forwarding algorithms (Baeza-Yates et al., 2009; Cambazoglu, Varol, et al., 2010) that can obtain the result quality of the global ranking by forwarding queries to a subset of non-local data centers instead of all. Query forwarding algorithms are beyond the scope of this paper. In this work, we assume either no query forwarding or an oracle query forwarding algorithm that correctly predicts the non-local data centers that will contribute to the global top \( k \) set and forwards queries to only those data centers.

### 2.2. Document replication framework

Our focus in this work is specifically on document replication, which may lead to efficiency and effectiveness improvements in the above-mentioned search architecture. In our replication framework, the documents that are frequently requested by the queries originating from a particular region are replicated on the data center responsible for that region. This approach improves the search quality attained by the local data centers, assuming to a scenario where queries are not forwarded between data centers. This is because the overlap between the local and global top \( k \) sets is likely to increase as more non-local documents are replicated on the local data centers. The replication of documents also leads to improvements in the query processing efficiency, assuming a scenario where queries are forwarded between the data centers. This is because fewer queries need to be forwarded and hence savings can be achieved in average query response times and the query workload of the search engine. The former benefit is because the network latency overhead incurred by query forwarding is eliminated for some queries. The latter benefit is since queries are processed over a smaller portion of the index.

Obviously, increasing the replication amount has a negative impact on query processing times of local data centers due to the increase in their index sizes. One of the main goals of this paper is to observe the trade-off between replication and search performance to identify the best replication strategies. In Sections 4–6, we investigate the impact of replication on the search quality, average response time, and query workload, respectively.

In our document replication framework, we assume that the amount of compute resources made available to the search engine remains the same after replicating documents. We also assume that the resources are distributed among the data centers in proportion to their index sizes, as an attempt to preserve the relative query processing performances of data centers. These assumptions are necessary to draw sound conclusions about the query processing efficiency under document replication since query processing times depend on both index sizes and computing powers of data centers.

We constrain the allowed replication amount in two different ways. In the first approach, a global capacity constraint bounds the total size of the index obtained after document replication, over the entire search system. This approach requires redistribution of the available hardware, after replication, among the data centers to preserve the ratio between the index size and the amount of compute resources of each data center. In the second approach, a local capacity constraint bounds the replicated index size of each data center relative to its local index size. This approach does not require redistribution of the hardware because the ratio between the local and replicated index sizes is the same for all data centers.

In Fig. 2, we illustrate these two types of constraints by an example. Fig. 2a shows a distributed search engine with four data centers, each initially having a local index of varying size and a fixed amount of hardware. Fig. 2b and c shows the indexes created after replication. In Fig. 2b, the total index size in the system doubles after replication. In Fig. 2c, the index size of each data center individually doubles. We note that the distribution of compute resources among the data centers changes in Fig. 2b, whereas it remains the same in Fig. 2c.

### 2.3. Formal problem definition

We are given a set \( D = \{d_1, d_2, \ldots, d_N\} \) of \( N \) documents, a set \( Q = \{q_1, q_2, \ldots, q_M\} \) of \( M \) queries, and a set \( C = \{C_1, C_2, \ldots, C_K\} \) of \( K \) data centers. Every data center \( C_k \in C \) initially stores a disjoint set \( D_k \subseteq D \) of documents and serves to a disjoint set \( Q_k \subseteq Q \) of queries. Moreover, each document \( d_k \in D \) is associated with a space overhead \( s_j \) (e.g., the number of postings the document contributes to the index) and the frequency \( f_j \) with which the document appears in the relevant results of queries in \( Q \).

---

3 However, these algorithms cannot restrict the set of contacted non-local sites to only those that are guaranteed to contribute to the global top \( k \) result set.

4 If additional hardware is used in order to compensate the increase in the index size due to replication, the financial implications of this should be taken into account.
**Definition 1 (Document replication).** A document replication \( \Phi \) is a mapping from a document \( d_j \in \mathcal{D} \) to a set of data centers in \( \mathcal{C} - \{ C_i \} \), i.e., \( \Phi(d_j) \) denotes the set of non-local data centers on which document \( d_j \) is replicated.

For a given document replication \( \Phi \), let \( \mathcal{D}_\Phi \) denote the set of documents that are either initially stored or later replicated on \( C_i \), i.e.,

\[
\mathcal{D}_\Phi = \mathcal{D} \cup \{ d_j : C_i \in \Phi(d_j) \}. 
\]

Let \( \phi_j \) be the number of non-local data centers on which \( d_j \) is replicated, i.e., \( \phi_j = |\Phi(d_j)| \). Given this notation, we define the following two document replication problems. The first problem assumes a global capacity constraint on the replication amount while the second assumes that each data center has a local capacity constraint.

**Problem 1 (Replication under global capacity constraint (DR-G)).** Given sets \( \mathcal{C}, \mathcal{D}, \mathcal{Q} \), and a global replication capacity \( G \geq 0 \), find a document replication \( \Phi \) such that the total replication amount does not exceed \( G \), i.e.,

\[
\sum_{d_j \in \mathcal{D}_\Phi} s_j \phi_j \leq G, 
\]

while a given performance objective is optimized.

**Problem 2 (Replication under local capacity constraints (DR-L)).** Given sets \( \mathcal{C}, \mathcal{D}, \mathcal{Q} \), and a local replication capacity \( L_i \geq 0 \) for each data center \( C_i \in \mathcal{C} \), find a document replication \( \Phi \) such that the replication amount on each \( C_i \in \mathcal{C} \) does not exceed its capacity \( L_i \), i.e.,

\[
\sum_{d_j \in \mathcal{D}_\Phi - \mathcal{D}_i} s_j \leq L_i, \quad \forall C_i \in \mathcal{C}, 
\]

while a given performance objective is optimized.

---

**Fig. 2.** A search engine with four data centers. Indexes: (a) before replication, (b) after replication under a constraint on the total index size (doubles the total index size), (c) after replication under a constraint on individual index sizes of data centers (doubles the index size of each data center).
2.4. Baseline solutions

As a simple baseline solution for the DR-G problem, we use a modified version of the document replication heuristic used in (Cambazoglu, Varol, et al., 2010). In this heuristic, each document $d_j$ is assigned a profit value $p_j$ estimated by the ratio between the past access frequency and the space overhead of $d_j$, i.e., $p_j = f_j/s_j$. The heuristic then iterates over all documents in decreasing order of their profits. At each iteration, a document $d_j$ is inserted into an initially empty set $G$ if the global capacity check $\sum_{d_k \in G} s_k (K - 1) \leq C_0$ does not fail. After the algorithm iterates on all documents, the documents in $G$ are replicated on all non-local data centers. We note that this heuristic slightly improves over a heuristic that terminates when the capacity check first fails (Cambazoglu, Varol, et al., 2010). The complexity of this algorithm is $O(N\lg N + NK)$, where the former term is the cost of sorting the documents and the latter is the cost of decoding the document replication solution.

For the DR-L problem, we use a baseline similar to the above-mentioned baseline. The sorted list of documents is traversed once, performing for each document $K$ separate checks on local capacities of data centers. Given a data center $C_i$, only the documents that are non-local for $C_i$ (i.e., those in $D - D_i$) are considered for replication on $C_i$. Documents are inserted into a separate set $L_i$ if the local capacity check $\sum_{d_k \in L_i} s_k \leq L_i$ does not fail for data center $C_i$. Once the traversal terminates, documents in each set $L_i$ are replicated on the respective data center $C_i$. The complexity of this algorithm is $O(N\lg N + NK)$, similar to the DR-G problem.

3. Setup

For simulations, we create two different setups, referred to as Europe and World. Both setups simulate a geographically distributed search engine with five data centers. In Europe, data centers are located in Germany, Spain, France, Italy, and UK. In World, they are located in Australia, Brazil, Canada, Germany, and Mexico. The former simulates an architecture with low network latencies between data centers while the data centers in the latter setup have high network latencies. We assume that each data center is located in the capital city of the respective country. We also assume that queries are issued from the five most populated cities in each country.

We predict the network latencies between the data centers (also, between the data centers and their users), using the speed of light on copper wire (200,000 km/s) and the great-circle distance between data center locations. We then project the predicted latencies to more accurate values by a formula obtained through regression between predicted and real-life latency measurements (Cambazoglu, Varol, et al., 2010). The real-life latency values are obtained over several geographically distant computers available to us.

The document collection contains about 200 million web pages crawled from the Web in 2009. This is a high-quality collection obtained after various cleansing (e.g., spam filtering). Using a proprietary classifier, we determine an initial local assignment for every document. We limit our study to only the documents that are assigned to one of the selected countries. We built separate local and replicated indexes, using Terrier (Onis et al., 2005).

For each data center, we extract samples with about 8.5 M and 7.0 M queries from the query logs of Yahoo!, for the Europe and World setups, respectively. We preprocess queries in four steps: query terms are case-folded, stop-words are eliminated, duplicate terms are unique, and query terms are alphabetically sorted. Finally, the queries are sorted in increasing order of arrival timestamps and split into four pieces. The first three pieces form the training set. The final piece forms the test set.

For both the training and test sets, we assume that the top $k$ result sets obtained over the entire index form the relevant documents. We evaluate the queries using a modified version of Terrier. The top $k$ result sets of training queries are used as inputs to the replication algorithms given in later sections. The top $k$ result sets of test queries are used for evaluation.

According to Fig. 3, the occurrence frequencies of the documents in the top $k$ search results follow a power-law distribution for both setups. In the Europe setup, the fraction of documents that appear in the top $k$ result set of at least one query is 0.53 for the training query set and 0.32 for the test query set. For the World setup, the fractions are 0.61 and 0.40, respectively.

In our simulations, we assume a setting where each node in a search cluster builds an index on three million documents. We estimate the total number of processors available to the entire search system based on the same assumption. We assign each data center a number of processors proportional to its index size. Hence, query processing times are comparable for data centers. During our simulations, we assume that the indexes are maintained in the main memories of the search nodes. The simulator assumes that the query processing cost is linearly proportional with the total number of postings associated with the query terms. We set the time cost of processing a single posting to 200 ns, which is an empirical value obtained from Terrier. We also assume a 20 ms preprocessing overhead per query. We omit all other costs as they are relatively less important, especially for low $k$ and $K$ values.

Regarding caching of previous search results, we conduct our experiments under two different scenarios: no result cache or an infinite result cache (Cambazoglu, Junqueira, et al., 2010a). In the latter scenario, we assume that each data center maintains a result cache that stores the results of all queries that are previously issued from its region. In either scenario,
the results are reported over the entire (both hit and miss) set of test queries. The result cache hit ratios are 0.56 (0.54) for the training query set and 0.51 (0.47) for the test query set in case of the Europe (World) setup.

In all plots, we use the tags DR-G and DR-L to denote the proposed replication algorithms, which use a global capacity constraint or local capacity constraints, respectively. The baselines given in Section 2.4 are similarly named as B-G and B-L. The default case with no replication is denoted by NR. We denote the scenario where the entire index is replicated on all data centers by FR. The replication amounts reported in the plots represent the percent replication on the entire system with respect to the replication amount in the FR scenario, i.e., \((K - 1)\sum_{q_i \in Q} s_i\). In case of DR-G, 100% replication corresponds to the FR scenario. We note that, in case of DR-L, the final replication amount on a data center may be lower than the local replication capacity if the index becomes fully replicated on that data center before the local replication capacity is reached. A single simulation run takes under an hour with the parameters in our experiments.

4. Optimizing search quality

4.1. Objective

Herein, we focus on replication algorithms for a specific search scenario where queries are processed only in their local data centers without any forwarding to non-local data centers. In this scenario, search result qualities may degrade as queries are processed over a subset of the entire index. The main idea behind our replication algorithms is to replicate, in a particular data center, the non-local documents that are frequently accessed by the users of that data center. As we replicate more documents, the search quality achieved by processing queries only within the local data centers is expected to be closer to that of a centralized search architecture.

We denote by \(R_i\) the set of relevant documents obtained by evaluating query \(q_i\) on an index built over the entire document collection \(D\). Also, we denote by \(C_i\) the data center that serves query \(q_i\) and by \(D_j^\Phi\) the document collection on \(C_i\) after documents are replicated via some \(\Phi\). We measure the result quality of a query by its precision, defined as follows.

Definition 2 (Precision of a query). For a given document replication \(\Phi\), the precision \(\rho(q_i, \Phi)\) of a query \(q_i\) is defined as the fraction of the relevant documents on \(C_i\) to all relevant documents, i.e.,

\[
\rho(q_i, \Phi) = \frac{|D_j^\Phi \cap R_i|}{|R_i|},
\]

The total precision \(P(Q, \Phi)\) of a document replication \(\Phi\) is defined as a sum over the precisions of individual queries in \(Q\), i.e.,

\[
P(Q, \Phi) = \sum_{q_i \in Q} \rho(q_i, \Phi).
\]

Given this definition, the problem is to find a feasible document replication \(\Phi\) that maximizes \(P(Q, \Phi)\) as the objective of the DR-G and DR-L problems.

4.2. Solution

Our solution to the DR-G problem is based on a combinatorial reduction to the well-known 0-1 knapsack problem (Cormen, Leiserson, Rivest, & Stein, 2009), where we are given a capacity \(W\) and a set \(T\) of \(n\) items, each associated with
a positive weight and a positive value. The goal is to find a subset $T' \subseteq T$ such that the total weight of items in $T'$ does not exceed $W$ and their total value is maximized. For every pair of document $d_j \in D$ and data center $C_i \in C$ such that $d_j \notin D_i$, we introduce an item $t_{ji}$ into the item set $T$ with an associated weight $w_{ji} = s_j$ and value $v_{ji} = \sum_{r \in R} (1/|R|)$. The knapsack capacity is set to the global replication capacity $G$. Based on the solution set $T'$ of the knapsack problem, we form the set $\Phi(d_j)$ of data centers where document $d_j$ will be replicated as

$$\Phi(d_j) = \{C_i : t_{ji} \in T'\}.$$ (6)

Here, each item $t_{ji} \in T'$ represents the replication of document $d_j$ on data center $C_i$. This replication increases the total precision by $1/|R|$ for each query $q_i \in Q$, such that $d_j \in R_i$, i.e., by $v_{ji}$. Replicating $d_j$ on $C_i$ consumes a space of $s_j$ from the available space, bounded by the global capacity $G$, i.e., we pick an item of weight $w_{ji}$ without exceeding the knapsack capacity $W$. Hence, the proposed reduction correctly maximizes $P(Q, \Phi)$.

For the DR-L problem, we use a slight variation of the above solution. Since each data center has its own local replication capacity, we solve a different knapsack problem instance for each data center $C_i$. Hence, the proposed reduction correctly maximizes $P(Q, \Phi)$.

Since the 0–1 knapsack problem is NP-hard, we use a greedy approximation algorithm (Cormen et al., 2009) in our solution. The algorithm has a complexity of $O(n \log n)$, where $n$ refers to the number of items in the knapsack instance. In the algorithm, each item is assigned a profit, set to the ratio between the item’s value and weight. The heuristic iterates over all items in decreasing order of profits. At each iteration, an item is placed in the solution set if its placement does not violate the capacity constraint.

Our solutions to the DR-G and DR-L problems have time complexities $O(Mk + NK\log(NK))$ and $O(Mk + NK\log(N))$, respectively. In these complexities, the first term is the total cost of encoding the document replication instance as knapsack instance(s). The second term is the cost of the greedy approximation algorithm as we solve one knapsack instance with $O(NK)$ items in the DR-G problem and $K$ knapsack instances each with $O(N)$ items in the DR-L problem. Since the cost of decoding the knapsack solution(s) as a document replication is $O(NK)$, it is not shown in the complexities.

4.3. Performance evaluation

Fig. 4 shows the average precision values observed as the replication amount increases. The average precision over a test query log $Q'$ is computed as $P(Q', \Phi)/|Q'|$. For a better visibility of the curves, the average precision values for NR (0.61 and 0.55, for Europe and World, respectively) are not displayed in the figure. Relative to these values, even with only 1% replication, considerable improvement in precision is observed (up to about 0.26 precision increase in either setup). When 16% of the index is replicated, the loss in the average precision is at most 10% with respect to the FR scenario, which naturally achieves a precision of one. This behavior is mainly due to the power-law distribution in the occurrence frequencies of documents in search results (previously shown in Fig. 3).
According to Fig. 4, both DR-G and DR-L achieve slightly better precision values than their respective baselines. As expected, the improvements over the baselines are less pronounced at high replication amounts. In general, DR-G has a relatively better performance than DR-L since DR-G has a larger solution space due to the flexibility of reassigning the hardware. The precision values in the World setup are lower than those in the Europe setup since the search results in the former setup contain relatively more unique documents. Hence, the World setup requires larger amounts of replication to achieve the same performance with the Europe setup. For example, achieving a precision of 0.92 requires more than 8% replication in World, whereas it requires only 4% replication in Europe (assuming DR-G).

5. Optimizing response time

5.1. Objective

In this section, we consider a search architecture where queries are forwarded between the data centers to retrieve all relevant documents so that there is no loss in the search quality, i.e., the precision is always one. We assume an oracle algorithm that forwards the queries to only the non-local data centers that contain relevant documents. As the performance objective in replicating documents, we try to reduce the average query response time.

In this scenario, the average query response time is determined by the time needed to compute the query results on different indexes and the network latencies between the data centers. The main idea in the proposed replication algorithm is to maximize the number of queries that can be entirely processed by the local data centers without any forwarding, thus eliminating the network latency and the overhead of query processing on non-local data centers. If all relevant results of a query are found in the local data center, the query is dubbed local. The locality of a query is defined as follows.

**Definition 3 (Locality of a query).** For a given document replication \( \Phi \), the locality \( \gamma(q_i, \Phi) \) of a query \( q_i \) is defined as

\[
\gamma(q_i, \Phi) = \begin{cases} 
1, & \text{if } R_i \subseteq \tilde{D}_i^\Phi \\
0, & \text{otherwise.}
\end{cases}
\]  

(8)

The total locality \( \Gamma(Q, \Phi) \) of a document replication \( \Phi \) is defined as a sum over the locality values of queries in \( Q \), after the documents are replicated via \( \Phi \), i.e.,

\[
\Gamma(Q, \Phi) = \sum_{q_i \in Q} \gamma(q_i, \Phi).
\]  

(9)

The goal is now to find a feasible document replication \( \Phi \) that maximizes \( \Gamma(Q, \Phi) \) as the objective of the DR-G and DR-L problems.

5.2. Solution

We formulate the problem as a variation of the set union knapsack problem (Kellerer, Pferschy, & Pisinger, 2004). In our problem, we are given a main set \( T \) of \( n \) items, where each item is associated with a positive weight. We are also given a family \( F = \{S_1, S_2, \ldots, S_m\} \) of \( m \) sets, where each set \( S_i \) is a subset of the main set, i.e., \( S_i \subseteq T \). The objective of the problem is to find a subset \( T' \) of the main set \( T \) such that \( | \{ S_i \in F : S_i \subseteq T' \} | \) is maximized while the total weight of the items in \( T' \) does not exceed a given capacity \( W \).

Since our problem is NP-hard, we employ a heuristic solution (Ntoulas & Cho, 2007). This heuristic follows the maximum benefit per unit cost policy (Cormen et al., 2009) and has no guarantee of optimality. In this heuristic, each item in \( T \) is assigned a profit value which is set to the ratio between the number of sets that contain the item and the item’s weight. The heuristic then iterates over all items in decreasing order of their profits. At each iteration, an item is placed in \( T' \) if the capacity constraint is not violated by the placement of the item. The heuristic has a complexity of \( O(m + n + n \log n) \), where \( n \) denotes the sum of the sizes of the sets in \( F \), i.e., \( n = \sum |S_i| \). The first two complexity terms refer to the cost of computing the profit values and the last term refers to the cost of sorting the items.

The solution we propose for the DR-G problem is based on a combinatorial reduction to our variation of the set union knapsack problem. Let us consider every data center \( C_i \in C \). For each document \( d_j \in D - D_i \), we introduce an item \( t_{ij} \) into the main set \( T \) with an associated weight \( w_{ij} = s_{ij} \). Moreover, for each query \( q_i \in Q \), we introduce a set \( S_i = \{ t_{ij} : d_j \in R_i - D_i \} \) into family \( F \). The capacity \( W \) is set to the global replication capacity \( G \). After obtaining solution \( T' \), we form the document replication \( \Phi \) as

\[
\Phi(d_j) = \{ C_i : t_{ij} \in T' \}.
\]  

Here, each item \( t_{ij} \in T' \) represents the replication of document \( d_j \) on data center \( C_i \). Thus, each set \( S_i \subseteq T' \) implies \( R_i \subseteq D_i^\Phi \) for the corresponding query \( q_i \in Q \). Due to (8) and (9), the objective of the reduced problem corresponds to maximizing the total

---

locality $\Gamma(Q, \Phi)$. Replicating $d_i$ on data center $C_j$ consumes a space of $s_i$ from the available space, bounded by the global capacity $G$, i.e., we pick an item of weight $w_i$ without exceeding the capacity $W$. Hence, the proposed formulation is correct.

For the DR-L problem, we use a minor variant of the above solution. Since each data center has its own local replication capacity, we solve a separate problem instance for each data center $C_i \in C$. For each document $d_i \in D_1$, we introduce an item $t_j$ into the main set $T'$ with the same weight as before. For each query $q_j \in Q$, we introduce a set $S_i = \{t_i : d_i \in \mathcal{R}_i \}$ into family $\mathcal{F}_j$. The capacity $W_i$ is set to the local replication capacity $L_i$. We then solve the problem instance associated with each data center $T'_i$ and obtain a solution set $T'_j$. After we obtain all $K$ solution sets, we form a document replication $\Phi$ as

$$\Phi(d_i) = \{ C_j : t_j \in T'_j \}. \quad \text{(11)}$$

We omit a discussion on the correctness of this solution as it is very similar to the above discussion on the correctness of our solution for the DR-G problem.

For both the DR-G and DR-L problems, the cost of encoding the document replication instance is $O(NK + Mk)$. For the DR-G problem, the complexity of the employed heuristic is $O(Mk + NK\lg(NK))$ while, for the DR-L problem, the complexity of solving the $K$ problem instances is $O(NK + NK\lg(NK))$. In both problems, the decoding of the obtained solution(s) has $O(NK)$-time complexity. Hence, our solutions to the DR-G and DR-L problems have overall time complexities of $O(Mk + NK\lg(NK))$ and $O(Mk + NK\lg(NK))$, respectively.

5.3. Performance evaluation

In Fig. 5, the average response times are shown for varying replication amounts, assuming that the previous search results are not cached in the data centers. These response time values are obtained using the simulation setup described in Section 3. They include the cost of processing queries over the indexes in the local and non-local data centers as well as the round-trip network latencies between the data centers and the network latencies between users and their local data centers (Cambazoglu, Varol, et al., 2010).

According to the figure, the best-performing replication strategy is **DR-G**, especially at low replication rates. In the **World** setup, at 16% replication, it outperforms its respective baseline **B-G** by about 5%. The performance gap between **DR-L** and **B-L** is relatively small.

The fraction of queries that can be fully answered by the local data centers increases as the replication amount increases (see Fig. 6). Hence, replicating more documents implies savings in network latencies. On the other hand, replication leads to an increase in local index sizes, implying an increase in query processing times. Consequently, the lowest average response times are observed with replication amounts between no replication and full replication. The optimum replication amount depends on the setup. In **Europe**, the lowest average response time is attained by **DR-G** with replication amounts in the 2-4% range, achieving about 17% reduction over the average response time attained by **FR** (117 ms versus 141 ms). However, as the replication amount further increases, the average response times start to increase. Beyond 32% replication, all strategies result in average response times higher than that of **FR**. For the **World** setup, the lowest response times are achieved by **DR-G** at relatively larger replication amounts. At 16% replication, the average response time saving relative to **FR** is about 19%. Interestingly, the performance of the replication strategies relative to **FR** deteriorates if the replication amount goes below 1%.

It is interesting to observe that, in the **Europe** setup, **FR** achieves a considerably lower average response time than **FR** (141 ms versus 206 ms), whereas **FR** performs better than **FR** in the **World** setup (209 ms versus 270 ms). This is because, in the **Europe** setup, query response times are dominated by the cost of evaluating queries over the index, whereas network latencies between the data centers form the main overhead in the **World** setup. Also, we observe that both the baseline and proposed strategies can outperform the **FR** and **FR** strategies when the replication amounts are low.

![Fig. 5. Average response time as the replication amount increases (no result cache).](image-url)
Fig. 7 shows the impact of using an infinite result cache on the average response times. As expected, there is a considerable reduction in response times as many queries can be readily served by the cache, without incurring any processing or network latency overheads. We observe that, in the Europe setup, the lowest response time is now achieved at slightly higher replication amounts. Similarly, in the World setup, more documents need to be replicated to perform better than the FR strategy.

6. Optimizing query workload

6.1. Objective

In this section, we continue to focus on a search architecture similar to the one described in Section 5, i.e., certain queries are processed on non-local data centers that are determined by an oracle query forwarding algorithm. The performance objective we consider is to reduce the query processing workload of the system. In particular, as a closely related optimization objective, we aim to minimize the average number of non-local data centers that participate in processing of queries by selectively replicating documents on data centers. We first provide some notation before presenting our solution.

**Definition 4** (Remote load of a query). For a given document replication $\Phi$, the remote load $\omega(q, \Phi)$ of a query $q_i$ is defined as the number of non-local data centers $C_r \in \mathcal{C} - \{C_i\}$ that contain at least one relevant document not available on local data center $C_i$, i.e.,

$$
\omega(q_i, \Phi) = \left| \{C_r \in \mathcal{C} : (D_r \cap R_i) \not\subseteq D_i^\Phi \} \right|.
$$

The total remote load of a document replication $\Phi$ is defined as a sum over the remote loads of queries in a $\mathcal{Q}$, after the documents are replicated via $\Phi$, i.e.,
The goal now becomes to find a feasible document replication \( \Phi \) that minimizes \( \Omega(Q, \Phi) \) as the objective of the DR-G and DR-L problems.

6.2. Solution

Our solution to the DR-G problem is based on a combinatorial reduction to the adapted version of the set union knapsack problem (see Section 5.2). For each pair of document \( d_j \in D \) and data center \( C_i \in C \) such that \( d_j \in D - D_i \), we introduce an item \( t_{ji} \) in the main set \( T \) with an associated weight \( w_{ji} = s_j \). Moreover, for each pair of query \( q_i \in Q \) and non-local data center \( C_i \in C - \{C_i\} \), we introduce a set \( S_{pi} = \{t_{ji} : d_j \in D_i \cap R_i, C_i = \hat{C}_i\} \) into family \( F \). The capacity \( W \) is set to the global replication capacity \( G \). After having a solution \( T^* \) to the reduced problem, we form the document replication \( \Phi \) as

\[
\Phi(d_j) = \{C_i : t_{ji} \in T^*\}.
\]

In this formulation, each item \( t_{ji} \in T^* \) represents the replication of document \( d_j \) on data center \( C_i \). Thus, each set \( S_{pi} \subseteq T^* \) implies \( (D_i \cap R_i) \subseteq D_j^* \) for query \( q_i \in Q \). This, in turn, implies that \( q_i \) is not forwarded to \( C_i \), decreasing by 1 the remote workload \( W(q_i, \Phi) \) incurred by \( q_i \). Due to (13), the objective of the reduced problem correctly captures the minimization of the total remote load \( \Omega(Q, \Phi) \). We omit a discussion on the capacity constraint since it is similar to the formulations in Sections 4 and 5.

For the DR-L problem, we use a modified version of the above solution. Because each data center has its own local replication capacity, we solve a separate problem for each data center \( C_i \). For each document \( d_j \in D - D_i \), we introduce an item \( t_{ji} \) in \( T_i \) with an associated weight \( w_{ji} = s_j \). For each pair of query \( q_i \in Q \) and non-local data center \( C_i \in C - \{C_i\} \), we introduce a set \( S_{pi} = \{t_{ji} : d_j \in D_i \cap R_i\} \). The capacity \( W_i \) is set to the local replication capacity \( L_i \). After having every solution set \( T_i \) associated with each data center \( C_i \), we form the document replication \( \Phi \) as

\[
\Phi(d_j) = \{C_i : t_{ji} \in T_i\}.
\]

For both the DR-G and DR-L problems, the encoding of the solution(s) has a complexity of \( O(NK + MK + MK) \). The complexity of the heuristic(s) for DR-G and DR-L are \( O(MK + MK + NKlgn(Nk)) \) and \( O(MK + MK + NKlgn(Nk)) \), respectively. In both problems, the decoding of the obtained solution(s) has \( O(NK) \)-time complexity. Hence, overall, the solutions to the DR-G and DR-L problems have time complexities of \( O(MK + MK + NKlgn(Nk)) \) and \( O(MK + MK + NKlgn(Nk)) \), respectively.

We note that although the constructed problem instances are different in case of optimizing the average response time and optimizing the workload, the solutions yield exactly the same replication pattern. This is because an item appears the same number of times among the sets and the heuristic takes into account only this number to decide on the replication pattern.

6.3. Performance evaluation

The workload values reported in this section are obtained by assuming that the workload incurred by a query on a data center is proportional to the size of the posting lists that need to be processed in that data center. The values are normalized by the workload estimated for query evaluation over the entire index. Hence, the normalized workload is 1.0 for the FR strategy.

![Fig. 8.](image-url) Normalized workload as the replication amount increases (no result cache).
Fig. 8 shows the normalized query workloads as the replication amount varies. According to the figure, with very little replication (e.g., 1%), it is possible to obtain large savings in the query workload. These large savings are mainly due to the sharp decrease in the average number of data centers that are active in query processing (see Fig. 9). However, as we replicate more documents, the local index sizes start to increase, diminishing the gains obtained by avoiding non-local computation.

In general, the proposed algorithms perform similar to the baseline algorithms. In the Europe setup, we observe that it is possible to decrease the workload by 21% (with 2% replication) relative to NR. In the World setup, the workload gain relative to NR goes up to 24% (with 4% replication). In both setups, it is possible to reduce the workload relative to FR by more than a half, replicating only a small fraction of the documents (e.g., 1%).

Fig. 10 shows the normalized query workload values assuming that an infinite result cache is deployed in data centers. In general, the performance behavior is not affected much by the presence of the result cache. On the other hand, the result cache has a major impact on the query workload. In case of the FR strategy, result caching leads to workload reductions of about 48% and 41% in the Europe and World setups, respectively (these values are not displayed in the plots).

7. Impact of query forwarding

In Sections 5 and 6, where we tried to optimize the average query response time and the query workload, respectively, we assumed that certain queries are forwarded between the data centers via an oracle query forwarding algorithm. However, in Section 4, where we tried to optimize the search quality, we assumed that queries are not forwarded between the data centers. Hence, in this section, we separately investigate the impact of query forwarding on the trade-off between the search quality and the average query response time.

Fig. 11 demonstrates the relative performance in precision and query response time values if we add query forwarding capability to a search system where the queries are originally not forwarded. In the figure, the values on the x axes of the
two plots show the reciprocal of the average precision values attained by the search system where no queries are forwarded. The values on the y axes show the response times for the system with oracle query forwarding capability. These values are normalized by the respective response times observed on the system that has no query forwarding capability. For document replication, we use the respective heuristics proposed for the DR-G problem. The data points in the plots correspond to particular replication amounts (left to right: 100%, 64%, 32%, 16%, 8%, 4%, 2%, 1%, and 0%).

As expected, when queries are forwarded, we observe gains in precision in exchange of an increase in average query response time. In particular, in the World setup, when the replication amount is 1% (the second data point from the right), the average response time almost doubles in exchange of about 25% increase in precision. In the same setup, achieving precision improvements similar to those in the Europe setup leads to relatively higher response time increase and higher document replication amounts. In general, result caching does not significantly affect the trade-off between search quality and response time, except for very low replication amounts.

8. Related work

There has been much research on distributed web search engines (Baeza-Yates, Castillo, Junqueira, Plachouras, & Silvestri, 2007; Barroso, Dean, & Hölzle, 2003; Cacheda, Carneiro, Plachouras, & Ounis, 2007a, Cacheda, Carneiro, Plachouras, & Ounis, 2007b; Orlando, Perego, & Silvestri, 2001). However, the feasibility of multi-site, geographically distributed search engines has not been studied until recently (Baeza-Yates et al., 2009; Cambazoglu et al., 2008, 2009, 2010). Cambazoglu et al. (2008) investigated the feasibility of geographically distributed web crawling. Reduced network latencies between data centers and web sites are shown to bring significant performance benefits for geographically distributed web crawling, compared to a centralized web crawling architecture. Baeza-Yates et al. (2009) conducted a study to demonstrate the performance gains in multi-site web search engine architectures. A cost model is developed and some simulation results are provided, illustrating the feasibility of multi-site web search. The same work also proposed a simple query forwarding algorithm that guarantees the correctness of the top k search results, relative to the results obtained by a centralized architecture. Cambazoglu et al. (2009) investigated the trade-off in search efficiency and effectiveness in multi-site web search architectures. An analytical cost model is developed to quantify the impact of replication and query forwarding on search efficiency. The same work also reported some results about the impact of geographically distributed web crawling on search quality, in particular, on the impact of web coverage on relevance. Using a setup similar to ours, Cambazoglu, Varol, et al. (2010) proposed a linear-programming-based query forwarding algorithm that improves the performance of the query forwarding algorithm proposed in (Baeza-Yates et al., 2009).

Both (Baeza-Yates et al., 2009; Cambazoglu, Varol, et al., 2010) employed simple replication algorithms in their performance evaluations. The former work prioritizes documents for replication according to their popularity in past search results. The latter work improves over the former work by also considering the space overheads of documents. We used a slightly modified version of the algorithm in (Cambazoglu, Varol, et al., 2010) as our baseline.

To our knowledge, so far, no work has proposed sophisticated document replication algorithms for multi-site search engines. However, replication is previously considered in many other contexts: content delivery networks (Kangasharju, Roberts, & Ross, 2002; Rabinovich, Rabinovich, Rajaraman, & Aggarwal, 1999), distributed database systems (Apers, 1998; Wolfson, Jajodia, & Huang, 1997; Deris, Abawajy, & Mamat, 2008; Loukopoulos & Ahmad, 2000; Plattner & Alonso, 2004), multimedia databases (Kwok, Karlapalem, Ahmed, & Pun, 1996), the Grid (Ranganathan & Foster, 2001, 2003; Lamehamedi,...
In this work, we investigated the problem of replicating documents on a multi-site, geographically distributed web search engine. We devised different heuristics aiming to improve various important performance criteria, such as search quality, average query response time, and query workload. The proposed heuristics are shown to slightly improve over the baseline replication algorithms used in previous works.

The main finding of our work is that replication can be a feasible alternative to fully partitioned or fully replicated multi-site search engines. We demonstrated that the optimum performance is achieved at certain replication amounts. We experimentally identified these optimum replication rates for different performance metrics and different search engine settings: the best replication rates are between 4% and 16% for minimizing the average response time and between 2% and 4% for minimizing the query workload. We also confirmed the performance benefits of result caching in multi-site search architectures.

A possible extension is to investigate the performance of our replication heuristics on related search architectures (e.g., P2P). Another research direction is to combine the replication problem with the user-to-site assignment problem, where the objective is to find a good static assignment between users and local data centers. Finally, the oracle query forwarding algorithm used in our work can be replaced with a practical query forwarding algorithm and the impact on the optimum replication rates can be observed.
Acknowledgments

This publication is based on work performed in the framework of the Project COST-ICT-248036, funded by the European Community, and partially supported by the Scientific and Technological Research Council of Turkey under grant EEEAG-109E019 and COST Action IC080 ComplexHPC.

References


Eray Özkural and Cevdet Aykanat are with Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey. E-mail: {erayo, aykanat}cs.bilkent.edu.tr.

B. Uçar is with LIP, ENS Lyon, 69364 Lyon, France. E-mail: bora.ucar@ens-lyon.fr.

Manuscript received 24 Dec. 2009; revised 7 July 2010; accepted 14 Oct. 2010; published online 18 Jan. 2011. Recommended for acceptance by K. Li.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-12-0666.

Digital Object Identifier no. 10.1109/TPDS.2011.32.

1632 IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 22, NO. 10, OCTOBER 2011

Parallel Frequent Item Set Mining with Selective Item Replication

Eray Özkural, Bora Uçar, and Cevdet Aykanat

Abstract—We introduce a transaction database distribution scheme that divides the frequent item set mining task in a top-down fashion. Our method operates on a graph where vertices correspond to frequent items and edges correspond to frequent item sets of size two. We show that partitioning this graph by a vertex separator is sufficient to decide a distribution of the items such that the subdatabases determined by the item distribution can be mined independently. This distribution entails an amount of data replication, which may be reduced by setting appropriate weights to vertices. The data distribution scheme is used in the design of two new parallel frequent item set mining algorithms. Both algorithms replicate the items that correspond to the separator. NoClique replicates the work induced by the separator and NoClique2 computes the same work collectively. Computational load balancing and minimization of redundant or collective work may be achieved by assigning appropriate load estimates to vertices. The experiments show favorable speedups on a system with small-to-medium number of processors for synthetic and real-world databases.

Index Terms—Parallel data mining, frequent item set mining, mining methods and algorithms, selective data replication, graph partitioning by vertex separator.

1 INTRODUCTION

1.1 Frequent Item Set Mining Problem

A transaction database consists of a multiset \( T = \{X|X \subseteq I\} \) of transactions. Each transaction is an item set, and it is drawn from a set \( I \) of all items. In practice, the number of items, \(|I|\), is in the order of magnitude of \( 10^6 \) or more. The number of transactions, \(|T|\), is usually larger than \( 10^{10} \). A pattern (or item set) is \( X \subseteq I \), any subset of \( I \), while the set of all patterns is \( 2^I \). The frequency function \( f(T, x) = |\{X \in T|X \subseteq x\}| \) computes the number of times a given item \( x \in I \) occurs in the transaction database \( T \), and it is extended to item sets as \( f(T, X) = |\{Y \in T|X \subseteq Y\}| \) to compute the frequency of a pattern. We use just \( f(x) \) or \( f(X) \) when \( T \) is clear from the context.

Frequent item set mining (FIM) is the discovery of patterns in a transaction database with a frequency of support threshold \( \epsilon \) and more. The set of all frequent patterns is \( \mathcal{F}(T, \epsilon) = \{X \in 2^I|f(T, X) \geq \epsilon\} \). We use just \( \mathcal{F} \) when \( T \) and \( \epsilon \) are clear from the context. In our algorithms, two sets require special consideration. \( \mathcal{F}_1 = \{x \in I|f(T, x) \geq \epsilon\} \) is the set of frequent items, and \( \mathcal{F}_2 = \{X \in \mathcal{F}||X| = 2\} \) is the set of frequent patterns with cardinality 2. In general, \( \mathcal{F}_k \) is the set of frequent patterns with cardinality \( k \). A significant property of FIM known as downward closure states that subsets of a frequent pattern are frequent, i.e., if \( X \in \mathcal{F}(T, \epsilon) \) then \( \forall Z \subseteq X, Z \in \mathcal{F}(T, \epsilon) \) [1].

If all items sets in \( \mathcal{F} \) are enumerated, the problem is known as the all FIM problem. Since the size of \( \mathcal{F} \) can be large, smaller enumeration problems have been defined such as closed [2] and maximal [3] FIM problems.

1.2 Related Work and Motivation

FIM comprises the core of several data mining algorithms, such as association rule mining and sequence mining. Frequent pattern discovery usually dominates the running time of these algorithms, therefore much research has been devoted to increasing the efficiency of this task. Since both the data size and the computational costs are large, parallel algorithms have been studied extensively [4], [5], [6], [7], [8], [9], [10], [11], [12]. FIM has become a challenge for parallel computing since it is a complex operation on huge databases requiring efficient and scalable algorithms.

While there are a host of advanced algorithms for parallel FIM, it is desirable to achieve better flexibility and efficiency. We have been inspired by the Partition algorithm [13] which divides the database horizontally and merges individual results, as well as Zaki et al.’s Par-Eclat algorithm [5] which redistributes the database into parts that can be mined independently. Also of immediate interest are the parallelizations of Apriori [1], most notably Candidate-Distribution [4] which pioneered independent mining. We ask the following questions. Can we design a parallel algorithm that exploits data-parallelism and task-parallelism? Can we find a model to optimize its performance? The present paper gives an affirmative answer to these questions by introducing an algorithm that divides the database into independently mined parts in a top-down fashion, according to an optimized distribution of the item set.

A review of related work with emphasis on parallelizations of Apriori and Par-Eclat may be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.32.
1.3 Contributions
We introduce two new coarse-grain data-parallel FIM algorithms using a top-down data partitioning scheme with selective replication. We propose a novel divide-and-conquer strategy suitable for parallelization of the FIM task. Our objective is to divide the whole transaction database into parts that can be mined independently. It turns out that we can distribute items so as to achieve our goal of independent mining, while replicating some items selectively, implying an amount of work that cannot be divided further in the same fashion. This optimization problem is cast as a Graph Partitioning by Vertex Separator (GPVS) problem where the partitioning objective corresponds to minimizing data replication or collective work (work that requires collective communication) by setting appropriate weights to vertices, and the partitioning constraint corresponds to maintaining storage balance or computational load by setting appropriate weights likewise. The transaction database distribution is independent of the underlying database representation and the serial mining algorithms employed. Experiments show that our method has competitive performance with respect to a state-of-the-art parallel mining implementation.

2 Transaction Database Distribution
In this section, we describe our theoretical contributions which will be developed into a parallel algorithm in Section 3. We make heavy use of the GPVS problem, which is briefly explained in the following.

The GPVS problem is to find a minimum weighted vertex separator $V_s$, removal of which decomposes a graph into components with roughly equal weights [14]. Let $G = (V, E)$ be a graph where $w(u)$ is the weight of vertex $u$. Let $w(U) = \sum_{u \in U} w(u)$ be the weight of a vertex set $U$. Let $Adj(u)$ denote the set of vertices that are adjacent to $u$, i.e., $Adj(u) = \{v | (u, v) \in E\}$. This operator can be extended to vertex sets by letting $Adj(U) = \bigcup_{u \in U} Adj(u) - U$.

Definition 1 (n-way GPVS). $\Pi_{VS}(G) = \{V_1, V_2, \ldots, V_n : V_i\}$ is a partition of the vertex set $V$ into $n + 1$ subsets $V_1, V_2, \ldots, V_n$ and $V_s$, such that for all $1 \leq i < j \leq n$ $Adj(V_i) \cap V_j = \emptyset$ (i.e., $Adj(V_i) \subseteq V_s$). The partitioning objective is to minimize $w(V_s)$. The partitioning constraint is, for all $1 \leq i \leq n$, $w(V_i) \approx [w(V) - w(V_s)]/n$ (parts have roughly the same weight).

The problem is NP-complete [15, ND 25 Minimum b-vertex separator]. A separator $V_s$ is said to be minimal if there is no subset of $V_s$ that is also a separator. The two-way GPVS will be denoted as $\Pi_{VS}(G) = \{A, B : S\}$.

We introduce a distribution method that can be used to divide the FIM task in a top-down fashion. The method operates on the graph $G_{F_2}$, which is defined as follows:

Definition 2. $G_{F_2}(T, \epsilon) = (F, F_2)$ is an undirected graph in which each vertex $u \in F$ is a frequent item and each edge $\{u, v\} \in F_2$ is a frequent pattern of length two, for a given database $T$ and support threshold $\epsilon$. The parameters $T$ and $\epsilon$ will be dropped when they are clear from the context.

We decode a two-way GPVS of the $G_{F_2}$ graph as a two-way distribution of the transaction database such that the two subdatabases obtained can be mined independently and therefore utilized for concurrency. In order for this property to hold, there is an amount of replication dictated by the vertex separator of $G_{F_2}$, which corresponds to the partitioning objective of GPVS. In the following, we first present the optimization aspects of our transaction database distribution technique. Then, we expound on our GPVS model for two-way transaction database distribution. Afterwards, we discuss minimization of data replication, followed by minimization of collective work and load balancing in the GPVS model. We then extend the two-way distribution scheme to $n$-way (for $n$ processors). Last, we show that our method is applicable to maximal and closed FIM problems.

2.1 Optimizing Parallel Frequent Item Set Discovery
Our objective of transaction database distribution is to divide a transaction database such that each subdatabase can be mined independently, while not inflating the data prohibitively and keeping the computational load balanced across subdatabases. Once such a distribution is obtained, a coarse-grain parallel frequent item set mining algorithm similar to Par-Eclat can be designed. Par-Eclat consists of a redistribution phase and a following local mining phase with no communication [5]. We present two algorithms: NoClique features completely independent mining with no communication just like Par-Eclat, while NoClique2 has a collective phase in which the running time is minimized and the rest of mining is independent. Since some data mining tasks on the subdatabases are performed independently in either algorithm, our method may be classified as a data-parallel algorithm that adopts input data partitioning with replication. This input data partitioning induces a task partitioning according to the owner-computes rule [16, Section 3.2.2], which states that the process assigned a particular data item is responsible for all computation associated with it.

We show that GPVS on $G_{F_2}$ is sufficient to designate such a distribution on the transaction database. Our work assumes that $G_{F_2}$ is sparse, because GPVS may not be feasible on dense graphs. Note that a sparse $G_{F_2}$ does not necessarily require the input database to be sparse.

We may begin formulating a problem for the coarse-grain data-parallel frequent item set mining algorithm as follows: Let a database $T$ contain a smaller database $T_i$, $i$ is a subdatabase of database $T$ if and only if, for every transaction $X \in T_i$, there is a distinct transaction $Y \in T$ such that $X \subseteq Y$ (recall that $T$ and $T_i$ are multisets). We will denote this ordering relation with $T_i \prec T$. The input database $T$ is distributed to a number of processors such that each processor has a subdatabase of the original transaction database. We denote this distribution by $D(T) = \{T_i | T_i \prec T\}$, possibly with replication. Also, we require the union of frequent patterns discovered in individual processors to be the set of frequent patterns of the entire data, i.e., $F(T, \epsilon) = \bigcup_{T_i \prec T} F(T_i, \epsilon)$. We call this the independent mining condition for a distribution $D(T)$.

In the following optimization problem, $w(\cdot)$ is any sensible cost measure that relates to mining a database, e.g., computational work, data size:
The objective in (1) seeks to minimize the total amount of redundancy that the distribution \( D(T) \) entails. We subtract the cost of the entire database from the sum of costs of distributed subdatabases \( T_s \) to denote this. Equation (2) is the distribution condition which states that the transaction database is distributed in any fashion, e.g., transactionwise, itemwise, or hybrid. Equation (3) is the independent mining condition, which ensures that independent mining of the subdatabases yields the frequent patterns of the entire database. The balancing condition (4) ensures that all processors share the cost fairly. At this stage, we do not explicitly state whether we are minimizing data redundancy or parallel overhead. However, some amount of data replication is often necessary for the independent mining condition to hold.

We will now expose our particular item redistribution scheme using information in frequent item sets of length two, which can be easily computed in parallel like in the design of Par-Eclat [5]. First, we will show how we can satisfy the distribution and independent mining conditions by showing a two-way item distribution. We will then analyze the objective and the balancing condition, explaining how we can assign weights and achieve load balance so that it becomes an acceptable solution to the coarse-grain parallel FIM problem.

### 2.2 Two-Way Itemwise Transaction Database Distribution

\( G_{F} \) is relatively easy to compute with respect to the complexity of the whole mining task, and its computation is amenable to efficient parallelization. It contains information that can be used to predict computational properties. For instance, the maximal cliques in \( G_{F} \) give us potentially maximal patterns [5], which in turn can be used to achieve task parallelism. Our data decomposition method, on the other hand, does not require finding maximal cliques. Instead, we use the GPVS of \( G_{F} \), which allows us to define independent mining on the transaction database by finding a particular distribution of the item set \( I \). Our item distribution identifies the absence of cliques across two sets of items rather than enumerating all cliques as in [5].

We will start by observing the similarity of GPVS objectives to ours. It turns out that we can use a GPVS of \( G_{F} \) to satisfy the independent mining conditions and to optimize parallelism at the same time. FIM task can be decomposed into mining two itemwise projections of the transaction database using GPVS. We use the projection operator \( \pi \) to explicitly show the vertical projections.

**Definition 3.** A transaction database projected from \( T \) over a set of items \( X \) is \( \pi_X(T) = \{ Y \cap X \} \) where \( Y \) is a transaction in \( T \).

Recall that a two-way GPVS is denoted as \( \Pi_{VS}(G) = \{ A, B : S \} \) where \( S \) is the vertex separator; and \( A \) and \( B \) are vertex parts. GPVS of \( G_{F} \) corresponds to a certain two-way distribution \( \{ A \cup S, B \cup S \} \) of the item set \( I \). This distribution induces a two-way transaction set distribution as follows:

**Definition 4.** A two-way transaction database distribution \( D(T) = (T_1, T_2) \) is induced by \( \Pi_{VS}(G_{F}) = \{ A, B : S \} \), where \( T_1 = \pi_{A\cup S}(T) \) and \( T_2 = \pi_{B\cup S}(T) \).

We require \( S \) to be a minimal separator. If \( S \) were not minimal, since the cost induced by the separator is included in both projections, removing a vertex from the separator would decrease the parallel cost. For that reason, it is better to choose a minimal separator, in case the GPVS heuristic does not find one. Fig. 1 depicts a sample transaction database and its \( G_{F} \) graph. \( \Pi_{VS} \) of this graph and the transaction database distribution \( D(T) \) induced by \( \Pi_{VS} \) is illustrated in Fig. 2. In the following text, we show that mining the database parts separately results in complete FIM of the original transaction database \( T \) satisfying (3). The proofs are found in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.32.

**Lemma 1.** If there is a frequent pattern \( P \) in \( T \), then there is a corresponding clique in \( G_{F} \), with vertices corresponding to items in \( P \).

**Lemma 2 (NoClique).** There is no frequent pattern with items in both \( A \) and \( B \) parts of \( \Pi_{VS} = \{ A, B : S \} \) of \( G_{F} \).

**Theorem 1 (Independent Mining).** Independent discovery of frequent patterns in projected databases \( T_1 = \pi_{A\cup S}(T) \) and \( T_2 = \pi_{B\cup S}(T) \) results in discovery of all frequent patterns in \( T \).

Theorem 1 can be improved slightly to suggest a more efficient parallelization. The frequent item sets within \( S \) do not have to be mined redundantly.
Lemma 4 (Minimum Replication). GPVS of $G_{F_2}$ with item frequencies as vertex weights minimizes the amount of data replication.

Minimizing data replication is also correlated to minimizing the total volume of communication during database redistribution. If the database is to be provided from a central server, then both objective functions are identical. Moreover, for an initial random distribution of the database, we are minimizing the upper bound of total communication volume during the redistribution phase. Note that the GPVS model will maintain storage balance among processors due to the partitioning constraint.

2.4 Minimizing Collective Work

Here, we take a look at possible choices for $w(\cdot)$ to minimize collective work. If the computational work estimate for a projection over a set of items $X$ is in the form of a summation of individual load estimates $l(\cdot)$ for items:

$$w(\pi_X(T)) = \sum_{u \in X} l(u),$$

then the proposed GPVS model will minimize collective work instead of minimizing data replication. It will also balance computational load due to the partitioning constraint.

Estimating the computational load is nontrivial, since we cannot know in advance how many patterns are present in the data. However, we can reason about the potential number of item sets in the search space that the mining algorithm will need to traverse. Although every algorithm follows a different strategy for determining frequent patterns, a measure of the portion of the search space containing potentially frequent patterns gives us a good estimate as in [4], [5]. In our method, however, computing the maximal cliques in $G_{F_2}$ (like in [5]) will incur additional overhead. Therefore, we use simpler functions for load estimation such as the following:

$$w_1(\pi_X(T)) = \sum_{u \in X} f(u),$$

$$w_2(\pi_X(T)) = \sum_{u \in X} \left(\frac{d(u)}{2}\right),$$

$$w_3(\pi_X(T)) = \frac{1}{2} \sum_{(u,v) \in X^2} f(\{u, v\}).$$

For estimating computation time, we can use (6) which calculates the data size within the projection over a given item set $X$ in a fashion resembling [4]. Equation (7) does not take into account the actual complexity of the task. An alternative approximation, which is inexpensive, can be found in [5]. Equation (7) is based on Zaki et al.’s item set clustering [5] where $d(u)$ is the degree of vertex $u$ in $G_{F_2}$. This estimate is an upper bound on the number of potential frequent patterns of length 3 obtained by calculating the number of 2-combinations of patterns with length 2.

Naturally, more advanced load estimate methods can be used to improve the accuracy. An obvious choice among the simpler functions is the total frequency of $G_{F_2}$ edges that fall within a given item set $X$ which gives us (8). Although $w_3(\cdot)$ does not strictly conform to (5), it can be made so by evenly
distributing the weight of each edge among its incident vertices, which yields an approximation to (8). In our experiments, we have found that \( w_1(\cdot) \) performed better or as well as \( w_2(\cdot) \) and \( w_3(\cdot) \) perhaps because it tends to reduce both data and task overhead.

### 2.5 Extension to \( n \)-Way Distribution and Any Level \( k \) of Mining

We will now show means to extend two-way transaction database distribution to an \( n \)-way distribution \( D(T) = \{T_1, T_2, \ldots, T_n\} \), where the independent mining conditions are generalized in the obvious way. The two-way transaction database distribution can be applied recursively to divide the two projected databases. Since the resulting projected databases are transaction databases themselves, we can apply the same method to divide them further.

In order to distribute the derived databases, one must obtain the \( G_{F_2} \) of the two parts. This can be accomplished by simply running the same algorithm for the projected transaction database, however, this can be costly. In the following, we present facts that lead to an efficient computational scheme to calculate an \( n \)-way distribution directly over \( G_{F_1} \). By making use of this simple observation, we avoid constructing intermediate projected databases. There is no need to recompute \( F \) and \( G_{F_2} \), since they are already known as shown by the following lemma:

**Lemma 5 (\( G_{F_2} \) of a Projection).** For a given item set \( X \subseteq I \), \( G_{F_2}(\pi_X(T), \epsilon) \) is the subgraph of \( G_{F_1}(T, \epsilon) \) induced by the vertex set \( X \).

We thus observe that we do not need to construct intermediate databases to calculate the \( G_{F_2} \)'s of the sub-databases in \( D(T) \).

**Corollary 2 (Fast Recursive Distribution).** Regarding the distribution \( D(T) = \{\pi_{A,S}(T), \pi_{B,S}(T)\} \) induced by \( \Pi_{V,S}(G_{F_2}) = \{A, B : S\} \), \( t h e n \epsilon \quad G_{F_2}(\pi_{A,S}(T), \epsilon) \) and \( G_{F_2}(\pi_{B,S}(T), \epsilon) \) can be calculated as vertex-induced subgraphs of \( G_{F_1}(T, \epsilon) \) by vertex sets \( A \cup S \) and \( B \cup S \), respectively.

The simplest way to obtain an \( n \)-way distribution is to use an \( n \)-way GPVS directly. Independent mining results extend to the \( n \)-way case in an obvious fashion. Thus, we will not prove them separately. However, there are a few differences from the two-way case, which we will now portray. In an \( n \)-way GPVS \( \Pi_{V,S}(G_{F_2}) = \{V_1, V_2, \ldots, V_n : S\} \) of the \( G_{F_1} \) graph, we note that the projection of \( S \cup V_i \) will result in independent mining. Although \( S \) is a minimal separator (i.e., no subset of it is a separator), we observe that not all \( S \) need to be replicated in all parts. In general, a portion of \( S \) will have to be replicated on processor \( i \) (i.e., \( \text{Adj}(V_i) \cap S \)) which may in the worst case correspond to \( S \). This implies that an item in \( S \) may be replicated in a different number of projected databases than others in the resulting distribution. The \( n \)-way GPVS model does not encapsulate this fact. However, it is easier to implement with an \( n \)-way GPVS tool.

Our formulation is also applicable to levels higher than two in case \( G_{F_1} \) is too dense. We define a graph \( G_{F_k} \) of \( k \)-length frequent item sets as follows:

**Definition 5.** \( G_{F_k}(T, \epsilon) = (F, E) \) is an undirected graph in which each vertex \( u \in F \) is a frequent item. For each frequent item set \( X \) of length \( k \) in \( F \), we insert a clique of items in \( X \) into this graph, i.e., one edge for each length 2 support of \( X \).

This definition allows us to use all the relevant results with no modification. The extension of results is trivial and will not be detailed due to space considerations. However, one property is important:

**Lemma 6 (Sparsity of Higher Levels).** \( G_{F_{k+1}} \) is not denser than \( G_{F_k} \).

### 2.6 Maximal and Closed FIM Problems

Our method is applicable to both variations of the FIM problem that compute subsets of \( F \). In maximal FIM, no set that is a subset of a frequent item set is output [3], [17]. In closed FIM, no set that is a subset of a frequent item set is supported by the same transactions is output [2]. For instance, consider frequent item set \( X = \{a, b, c\} \). In maximal FIM, no subset of \( X \) like \( \{b, c\} \) will be output, and in closed FIM, \( \{b, c\} \) will be output if and only if it occurs in a different set of transactions than \( X \). After item distribution, if a processor has a set of frequent items \( X \), it also has all transactions belonging to all subsets of \( X \). Thus, both maximal and closed item set mining can be parallelized with our method.

### 3 Two Data-Parallel Algorithms

In this section, we present \( \text{NoClique} \) and \( \text{NoClique2} \), which are coarse-grain data-parallel algorithms based on the theoretical observations of Section 2. Our algorithms compute the set of frequent item sets and their frequencies for a given global transaction database \( T \) and a support threshold \( \epsilon \) on \( n \) processors. The implementation of \( \text{NoClique2} \) is built upon our new vertical serial FIM algorithm \( \text{Bitdrill} \).

#### 3.1 NoClique: The Black Box Parallelization

\( \text{NoClique} \) is a direct application of Theorem 1 and Corollary 2. First, we compute \( G_{F_1} \). Then, we recursively apply the two-way database distribution of Definition 4 until we have \( n \) parts, using fast recursive distribution (Corollary 2). For instance, assume \( n = 4 \). Consider the two-level partitioning that results in the \( G_{F_1} \) graphs of \( T_1 \) and \( T_2 \) in Fig. 2. We have parts \( A, B \), and separator \( S \) at the top level; we take two vertex-induced subgraphs of \( G_{F_1} \) over \( A \cup S \) and \( B \cup S \). If we apply GPVS recursively on \( G_{F_1}(T_1) \) and \( G_{F_1}(T_2) \), we can obtain four overlapping item sets that define an item distribution such as \( D(I) = \{\{b, c, f, g\}, \{b, e, g\}, \{a, d, e, g\}, \{d, e, h\}\} \). Now, each item set in \( D(I) \) can be assigned to a processor. The database is redistributed to processors according to this assignment. Afterwards, we can run any given sequential FIM algorithm on each processor simultaneously and independently, with no further communication. The main advantage of this parallelization is that any serial FIM algorithm that starts from level 3 can be used. The disadvantage is that, since some subgraphs of \( G_{F_1}(T) \) are replicated, there is some redundant work. Therefore, this algorithm is suitable only for sparse problem instances that do not require much replication. The recursive application of the two-way item distribution can be carried out in parallel, and of course it is much better if a parallel GPVS algorithm
can be used. We have obtained extremely high superlinear speedups in the parallelization of FP-Growth and AIM2 which prompted us to continue research in this direction. We applied NoClique to parallelize kDCI \[11\], \[18\], \[19\], LCM \[20\] (all FIM), DCI-Closed \[11\], AIM \[21\] (version 2), and FP-Growth-Tiny \[22\].

3.2 Bitdrill: Our Sequential Mining Algorithm

Bitdrill is a new efficient sequential FIM code that we developed as a basis for our NoClique2 algorithm. It uses tries (prefix trees) to store sets of item sets, where each item set is a string of items in decreasing order of frequency. It uses tidlists (a tidlist is a list of transaction ids an item occurs in) to store the database in memory; linked lists of items are used for sparse items and bit vectors are used for dense items. The algorithm proceeds in BFS order and affords fast candidate generation in a fashion similar to kDCI (which is one of the most efficient FIM algorithms together with LCM). We use a regular tree data structure instead of prefix arrays as in kDCI. Fast candidate generation relies on the fact that the prefix tree already captures much of the proximity between two item sets needed for generating a candidate. Let A and B be two frequent item sets of length k that share a prefix of length k−1. Both will be the children of the same internal node in the prefix tree. Thus, one can simply take their union and generate a (k+1)-length candidate item set. When we consider the Downward Closure lemma, we will see that all candidates can be generated in this fashion since any subset of a candidate must be frequent and will have frequent subsets with all possible (k−1)-length prefixes. Thus, we can simply traverse the prefix tree and generate all candidates by taking 2-combinations of the children of each internal tree node that corresponds to a (k−1)-length prefix. After the candidate is generated, it is subject to further pruning employing the Downward Closure lemma. Since we use a vertical representation, the frequency of candidates can be calculated on the fly. To speed up the tidlist intersections, we use a cache to hold all the tidlist intersections in the path to the root, so that a single additional intersection is sufficient to count the transactions in a candidate item set. The overall algorithm is quite efficient; its performance is comparable to kDCI for dense databases and is faster than kDCI for sparse databases (due to the dynamic tidlist representation).

3.3 NoClique2 Algorithm

3.3.1 Assumptions

We assume that the number of items is much greater than n (the number of processors). We assume that the database has already been mined up to level l and a GPVS of $F_{l+1}$ has been computed. In the following, we use k as a variable level and we start mining from level $l+1$. Our algorithm will work better when $F_{l+1}$ can be partitioned well. In many cases, there is a suitable l.

3.3.2 Overview

Using our n-way GPVS-based item distribution/replication scheme, we decompose the mining problem into a collective work phase (with communication), and independent work phase (with no communication) following the observations in Corollary 1. The algorithm takes as input at each processor a local transaction database $T_{local}$ and an absolute support threshold $\epsilon$. We assume that T has been partitioned transactionwise into $T_{local}$s prior to the execution of the mining algorithm. We also supply the set of frequent item sets up to and including level l, the graph $G_{l}$ corresponding to level l, and a heuristic GPVS solution $\Pi_{V_{S}}(G_{l})$. The algorithm is comprised of four phases:

1. Redistribute items with selective replication.
3. Mine nonreplicated items independently.
4. Merge frequent item sets across replicated and nonreplicated sets of items.

The phases of our algorithm are explained in the following.

3.3.3 Redistribution of Items

Items are distributed according to an n-way GPVS of $F_{l}$.

3.3.4 Mining Replicated Items in Parallel

Since each processor has the tidlists of all the items in $V_{r}$, we can parallelize candidate generation and testing steps fairly well, starting from level $l+1$. Assume that for a previous level $k$, we have the frequent item sets inserted in decreasing frequency order into a prefix tree. On the prefix tree, we can efficiently generate candidates for level $k+1$ using fast candidate generation of Bitdrill. While traversing an internal node for a $k-1$ length prefix during fast candidate generation (Section 3.2), for a children (all of which are leaves) at most $a^{2}$ candidates can be generated. Those internal nodes are each given the just mentioned upper bound of $a^{2}$ as weight and we partition the prefix tree into n subtrees of alphanumerically consecutive item sets, where each subtree has a roughly equal sum of weights. Each processor generates a distinct set of candidates with fast candidate generation on the assigned subtree, and then intersects tidlists to check their frequencies, simultaneously. At the end of the iteration, the (locally output) frequent item sets of length $k+1$ are gathered on all processors. The iteration continues until frequent item sets are exhausted. Since both candidate generation and testing steps are parallel, and the subtree-based distribution of candidates makes local tidlist caches useful, this phase works fairly fast.

3.3.5 Independent Mining

On each processor $i$, there is a distinct set of tidlists corresponding to items in $V_{r}$ not present on any other processor. The frequent item sets within $V_{r}$ are mined using a levelwise vertical mining algorithm (Bitdrill) starting from level $l+1$.

3.3.6 Merging Frequent Item Sets

As the last step, we mine frequent item sets that have items in both the replicated $V_{r}$ and the nonreplicated items $V_{i}$ (for
a processor $i$). We use the output of two preceding phases to achieve this. We start with level $l + 1$ again. For merging frequent item sets in a level $k + 1$, assume that we have the frequent item sets in level $k$. We use both the frequent items in level $k$ and the already mined frequent item sets in $V_s$ and $V_t$ to prune as many frequent item sets as possible. We apply the well-known Downward Closure pruning. Furthermore, any generated candidate must be combined from already-mined two sets of frequent items that we are merging. We have adapted fast candidate generation to work with our item set merging logic. We have achieved this in two complementary steps explained below.

**First step.** For any candidate item set that has at least 2 items in either part ($V_s$ or $V_t$), we can use ordinary fast candidate generation over the frequent item sets in level $k$ that have items in both $V_s$ and $V_t$ sets. After that, we check for a candidate $C$ if $C \cap V_s$ is frequent in the replicated database, which is the output of phase 2, and $C \cap V_t$ is frequent in independent database, which is the output of phase 3.

**Second step.** Consider a $(k + 1)$-length candidate $C$ with one item $x$ in one part and $k$ items in the other part. Not all of its $k$-length supports have at least one item in either part, therefore, $C$ cannot always be generated from the frequent item sets between parts in level $k$ with fast candidate generation. We make use of the observation that if $C$ is frequent, $V_s$ will have $k$-length subsets that include $x$. While traversing the $(k - 1)$ length prefixes in the prefix tree, for each item $x$, we construct a set of conditional $(k - 1)$-length patterns that have items in only one part by removing $x$. Then, for each item $x$, we generate $k$-length candidates from the corresponding set of conditional patterns using fast candidate generation. These $k$-length candidates have items in only one part and are checked if they are already frequent in that part. If so, then we add $x$ back to generate $C$ and apply the Downward Closure pruning restricted to $k$-length subsets across both parts.

After fast candidate generation, we use the ordinary caching and intersection routines of Bitdrill to calculate frequencies. Iteration continues until exhaustion of merged patterns. Note that this step can be used for any distribution of items, not just for our GPVS-based distribution.

### 3.4 Repl-Bitdrill Algorithm

The phase of mining replicated items in parallel can be considered as a stand-alone parallel FIM algorithm, which is similar to the second phase of ParDCI [10]. When used on its own, we call it Repl-Bitdrill as it replicates the tidlists of all frequent item sets on all processors, at the level that it starts mining. Note that NoClique2 degenerates to Repl-Bitdrill when partitioning is impossible, i.e., all items are replicated. Repl-Bitdrill is used in Section 4 to experimentally show the merits of partitioning in NoClique2.

### 3.5 Comparison with Par-Eclat

To put things in perspective, it may help to note the ancestry of our algorithm. Our algorithm is close to Par-Eclat [5]. The most important similarities between two algorithms are: 1) We use the same graph of two items when $l = 2$. 2) We also use graph theoretic observations to cluster items. 3) We also distribute items so that each processor mines independently with no further communication. On the other hand, we highlight the following differences:

#### 3.6 Implementation

Our implementations of Bitdrill, Repl-Bitdrill, NoClique, and NoClique2 are written in C++ using MPI. The computation of GPVS in NoClique2 is relevant to the experiments. We use the hypergraph partitioning-based formulation for computing a GPVS of $G_F$, [23], [24]. To that end, we use the hypergraph partitioner PaToH [25], [26].

### 3.7 Applicability to Dense Data

We have indicated that our algorithm is not supposed to work well with problem instances that give rise to a dense graph. In dense databases, this is not necessarily the case, and we have observed that our method works even with such databases. When the graph is quite dense, a large number of items are replicated, and our algorithm degenerates into an algorithm like the second phase of kDCI that replicates all items. Often, choosing a more suitable support threshold or a starting level for our algorithm helps.

### 4 Experiments

Here, we give a summary of our experiments; more detailed experimental results are given in Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.32. We have run our algorithms NoClique2 and Repl-Bitdrill as well as ParDCI on one synthetic (T60.110.2000K) and three real-world databases on a Beowulf cluster. In Table 1, NoClique2, Repl-Bitdrill, and ParDCI are abbreviated as NC2, RBD, and PDCI, respectively. As seen in Table 1, out of 16 parallel mining cases, NoClique2 achieves considerably higher speedup in eight cases, whereas NoClique2 and Repl-Bitdrill attain close speedups in eight cases. ParDCI achieves the highest speedup in two cases. For the tree database, ParDCI unfortunately crashed, and we could not measure its running time. We would expect it to have good performance as in user-likesmovies which is similarly dense. For the sparser database T60.110.2000K, NoClique2 achieves
better speedups than the other algorithms. Only NoClique2 attains increasing speedup with increasing number of processors for all the databases. Repl-Bitdrill and ParDCI show this nice property only for two databases each.

Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.32, contains a detailed explanation of the databases, experimental setup, speedup, partitioning quality, running time dissection, speedups of NoClique parallelizations, and discussion of observed superlinear speedups in NoClique. With regards to partitioning quality, we have examined expected versus actual load imbalance and data replication ratio. We have seen that our heuristic load estimates work but could be much improved. Data replication is controlled well enough but it is better for small number of processors. It turns out that in the sparse database, independent mining phase dominates and in the dense databases (user-likesmovies and trec) the collective work phase dominates. For these databases, the replication approach of Repl-Bitdrill and ParDCI is effective. However, in an important other case (trec.lp.20000) which represents the “long tail” in a real-world data set, there is a mixture of both phases, and ultimately NoClique does much better than Repl-Bitdrill and ParDCI, showing the true potential of our approach. The trec.lp.20000 database contains items in the trec with a frequency of 200,000 and lower. In the trec database, it is not possible to mine frequent item sets beyond a narrow set of items due to the power-law-like distribution of items, however in such real-world databases we are interested in relationships among a large number of items.

5 Conclusions

We have proposed an item distribution method that depends on theoretical observations that identify lack of cliques among two sets of items in $G_F$. The mining problem is decomposed into independent subproblems using a GPVS model which encapsulates the minimization of task or data redundancy as well as computational load or storage balance. We showed that this model can be extended to $n$-way distribution and any level of mining. Based on our distribution model, we designed and implemented two parallel FIM algorithms called NoClique and NoClique2. Experiments with synthetic and real-world databases on a Beowulf cluster showed considerable speedups, thus affirming the validity of our model.

A discussion of future work is present in Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.32.

Acknowledgments

This work is partially supported by The Scientific and Technological Research Council of Turkey under grant EEEAG-109E019 and COST Action IC080 ComplexHPC. Part of the experimental tests were carried out at the TUBITAK ULAKBIM High Performance Computing Center. The authors thank Claudio Luchesse for making ParDCI available to them. The authors thank Bart Goethals for providing the benchmark results of FIMI 2004 experiments. Thanks to anonymous reviewers for recommending many improvements.

References


Eray Özkural received the BSc and MSc degrees from the Computer Engineering Department of Bilkent University, Ankara, Turkey. He is currently working toward the PhD degree from Bilkent University. His research interests include parallel computing, data mining, information retrieval, algorithms, machine learning, algorithmic information theory, artificial general intelligence, and philosophy of mind. He has worked as a parallel computing researcher at the European Union SEE-GRID project and Erendiz Süperbilgisayar Ltd. Among his recent innovations is a memory system for artificial general intelligence.

Bora Uçar received the PhD degree in computer engineering from Bilkent University, Ankara, Turkey, in 2005. He worked as a postdoctoral researcher at the Mathematics and Computer Science Department of Emory University, Atlanta, and at the Parallel Algorithms Project, CERFACS, France. Since January 2009, he is conducting research activities at ENS Lyon, France, as a research scientist of CNRS. His research interests include combinatorial scientific computing, parallel and high-performance computing, and sparse matrix computations.

Cevdet Aykanat received the BS and MS degrees in electrical engineering from Middle East Technical University, Ankara, Turkey, and the PhD degree in electrical and computer engineering from Ohio State University, Columbus. He was a Fulbright scholar during his PhD studies. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor and the associate provost. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects, parallel computer graphics applications, parallel data mining, graph and hypergraph-theoretic models for load balancing, high-performance information retrieval systems, parallel and distributed databases, and grid computing. He has (co)authored about 60 technical papers published in academic journals indexed in ISI and his publications received above 400 citations in ISI indexes. He is the recipient of the 1995 Young Investigator award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science award. He was appointed as a member of IFIP Working Group 10.3 (Concurrent System Technology) in April 2004, as a member of the EU-INTAS Council of Scientists in June 2005, and as an associate editor of the IEEE Transactions of Parallel and Distributed Systems in December 2008.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.
ABSTRACT

Query forwarding is an important technique for preserving the result quality in distributed search engines where the index is geographically partitioned over multiple search sites. The key component in query forwarding is the thresholding algorithm by which the forwarding decisions are given. In this paper, we propose a linear-programming-based thresholding algorithm that significantly outperforms the current state-of-the-art in terms of achieved search efficiency values. Moreover, we evaluate a greedy heuristic for partial index replication and investigate the impact of result cache freshness on query forwarding performance. Finally, we present some optimizations that improve the performance further, under certain conditions. We evaluate the proposed techniques by simulations over a real-life setting, using a large query log and a document collection obtained from Yahoo!

Categories and Subject Descriptors
H.3.3 [Information Storage Systems]: Information Retrieval Systems

General Terms
Algorithms, Design, Performance, Experimentation

Keywords
Search engines, distributed IR, query forwarding, optimization, linear programming, index replication, result caching

1. BACKGROUND

Commercial web search engines of the past relied on a single search site (data center), which processed queries issued from all around the world. This approach had the typical scalability problems in centralized architectures. Moreover, queries issued from distant locations suffered from poor response times as the network latency between the user and the site became an issue. For such queries, either the query processing times had to be shortened, thus degrading the result quality, or users experienced unreasonable response times, which had implications on user satisfaction [16].

At this point, replicating the data (i.e., the web collection and the inverted index built upon it) over multiple, geographically distant search sites emerged as a feasible solution. In this strategy, each geographical region is mapped to a nearby search site. A search site processes over its full web index only the queries originating from the regions assigned to itself\(^1\). Although this strategy reduces network latencies, the scalability still remains as an issue since the entire web index had to be maintained on all search sites and queries are evaluated over the full web index.

A strategy that contrasts replication is to partition the data disjointly and assign each site only the documents obtained (crawled) from its region [8]. In this strategy, local queries of a region are evaluated over the partial index in the corresponding search site. The underlying assumption here is that users are interested more in documents located in their own region and local documents are more relevant for queries originating from the same region. As queries are now evaluated over small subsets of the web index, gains are possible in query processing time and throughput [8], along with other gains, such as those in web crawling [9].

Unfortunately, although the assumption about having high relevance between the documents and queries of the same region is reasonable, this is not true for all queries as some queries target non-regional documents [4]. This implies that evaluation over a partitioned index will lead to inferior search quality as some relevant, non-local documents are not retrieved. The problem of accessing non-local documents has two immediate solutions: taking the data to where it is sought and/or taking the queries to what they seek. The first is an offline solution that requires partial replication of the popular documents in a region on some non-local search sites. The second is an online solution that requires selective replication.

\(^1\)Herein, we refer to such queries as local queries.
forwarding of queries between search sites to extend coverage of search results. Our focus in this paper is on the latter approach, but we briefly touch to the former as well.

Selective query forwarding works as follows. The local search site receives a query and makes a decision about the quality of the locally computed results (relative to globally computed results, which would have been obtained through evaluation over the full index). If it is predicted that the local ranking misses some documents that would have appeared in the global ranking, a forecast is made about which search sites might have those documents. The query is then forwarded to those sites for further processing over non-local indexes and more results are retrieved. Finally, non-local and local results are merged and returned to the user.

The predictions made may lead to false positives (the query is forwarded to a site with no useful results, thus degrading performance) as well as false negatives (the query is not forwarded to a site with useful results, thus degrading the search quality). In this paper, our focus is on query forwarding techniques that preserve the search quality, i.e., those with no false negatives. This requires correctly identifying all search sites that will contribute to the global top $k$. In the mean time, the number of contacted sites with no useful results should be kept minimal as this has an impact on the performance and overall costs of the search engine.

A recent study [3] has proposed a thresholding technique that preserves the search quality while reducing the number of sites contacted. In this work, we build upon that work and propose a new thresholding algorithm that substantially improves the algorithm in [3] in terms of efficiency. Our algorithm has an offline phase, in which past user query logs are used to create offline queries, for which the maximum possible score attainable on each site is precomputed and globally replicated. In the online phase, this information is used in a linear programming (LP) formulation to set upper bounds on possible non-local site scores for new queries. Forwarding decisions are given based on comparisons between these bounds and the $k$th top score on the local site.

The following are the contributions of this paper:
- We describe an LP-based thresholding algorithm that significantly outperforms the current state-of-the-art [3].
- We evaluate a heuristic for partial index replication.
- We investigate the impact of result caching and cache freshness on query forwarding performance.
- We present several optimizations that provide further performance improvements under certain conditions.

The rest of the paper is organized as follows. Section 2 presents the considered geographically distributed search engine architecture and the associated query forwarding problem. We describe the proposed thresholding algorithm in Section 3. Experimental framework is given in Section 4. Section 5 provides the performance results. Further optimizations are proposed and evaluated in Section 6. Related work is surveyed in Section 7. We conclude in Section 8.

### 2. QUERY FORWARDING PROBLEM

#### 2.1 Architecture

We consider a distributed architecture with $N$ geographically distant search sites, where each site is assigned a nearby geographical region and is responsible for crawling and indexing only the documents in its assigned region. That is, the global web index is disjointly (document-based) parti-

![Figure 1: A geographically distributed search engine architecture with query forwarding.](image)

3Result sets are merged according to document scores. We assume that global collection statistics are available on all sites, and scores generated by different sites are compatible.
mize any form of redundancy and inefficiency incurred by forwarding decisions that do not improve the search quality.

### 2.3 Performance Metrics

Let \( Q_L \) and \( Q_F \) be the sets of locally processed and forwarded queries, respectively. Let \( F^q \) denote the set of non-local sites that query \( q \) is forwarded to. We employ two performance metrics: the fraction \( \alpha \) of locally processed queries

\[
\alpha = \frac{|Q_L|}{|Q_L| + |Q_F|},
\]

and the average number \( \beta \) of non-local sites hit per query

\[
\beta = \frac{\sum_{q \in Q_L \cup Q_F} |F^q|}{|Q_L| + |Q_F|}.
\]

In addition to these metrics, we measure the average query response time and the average query processing workload (relative to query processing over the full index) of the search engine. Since all of our optimizations are quality-preserving, herein, we do not use a result quality metric (e.g., P@N).

Let \( q \) denote a query submitted by some user \( u^q \) to a local search site \( S^q \), and let \( S_i \in F^q \), where \( 1 \leq i \leq |F^q| \). Also, let \( I \), \( I^q \), and \( \hat{I} \) denote the global index, the local index of \( S^q \), and the local index of \( S_i \), respectively. If a query is processed locally, there are mainly two cost components in the query response time\(^3\). The first cost (steps 1 and 6 in Fig. 1) is the user-to-site network latency \( \ell(u^q, S^q) \), which is incurred while \( q \) is transferred from \( u^q \) to \( S^q \) and also while the final results are transferred from \( S^q \) to \( u^q \). The second cost (step 2 in Fig. 1) is the computational cost \( t(q, \hat{I}) \) of processing \( q \) over \( \hat{I} \). The query response time then becomes

\[
T_L(q) = 2 \times \ell(u^q, S^q) + t(q, \hat{I}).
\]

If the query is forwarded, then there are two additional costs. The first cost (steps 3 and 5 in Fig. 1) is the site-to-site network latency \( \ell(S^q, S_i) \), which is incurred while \( q \) is transferred from \( S^q \) to \( S_i \) and also while non-local results are transferred from \( S_i \) to \( S^q \). The second cost (step 4 in Fig. 1) is the non-local query processing cost \( t(q, I_i) \), i.e., the cost of processing \( q \) remotely on \( I_i \). The response time becomes

\[
T_F(q) = T_L(q) + \max_{S_i \in F^q} \{2 \times \ell(S^q, S_i) + t(q, I_i)\}.
\]

In Eq. (4), we take the maximum of all remote response times since queries are transferred to non-local sites at the same instant and the highest remote response time determines the waiting time of the local site. We can now compute the average query response time \( T_{avg} \) as

\[
T_{avg} = \frac{\sum_{q \in Q_L} T_L(q) + \sum_{q \in Q_F} T_F(q)}{|Q_L| + |Q_F|}.\]

Let \( W(q, I) \) represent the workload\(^4\) incurred to the search engine when evaluating \( q \) over an index \( I \). For a given query set \( \tilde{Q} \), we compute the relative workload \( W_{rel} \) as the ratio of the total workload incurred in our architecture to the workload incurred by evaluation over the full index, i.e.,

\[
W_{rel} = \frac{\sum_{q \in \tilde{Q}} (W(q, I) + \sum_{S_i \in F^q} W(q, I_i))}{\sum_{q \in \tilde{Q}} W(q, \hat{I})}.
\]

\(^3\)We assume that result merging as well as various other costs are negligible, as this is the case for low \( k \) and \( N \) values.

\(^4\)Herein, we approximate the workload incurred by a query as the sum of inverted list lengths of all terms in the query. Interested readers may refer to [11] for other possibilities.

### 3. THRESHOLDING ALGORITHM

#### 3.1 Preliminaries

We assume that queries are processed over the inverted index in the AND mode [15], which is often the case in practice. In this mode, only the documents that contain all query terms are retrieved. The score of a document is computed simply by summing the term scores, indicating the relevance of the term to the document (e.g., BM25)\(^5\). Optionally, document-specific scores (e.g., PageRank) may be added to the final score. The technique proposed herein is applicable only to the former type of scoring. Extending it to cover the latter type requires further research.

As our aim is to preserve the search quality of a centralized architecture, a query \( q \) should be forwarded to any non-local site that would have at least one result in the global top \( k \) set. A non-local site \( \tilde{S} \) can contribute to this set only if the top score \( s(q, 1, \tilde{S}) \) it computes for \( q \) is larger than the \( k \)th score \( s(q, k, \tilde{S}) \) that local site \( \tilde{S} \) computes\(^6\). Obviously, it is not possible to know \( s(q, 1, \tilde{S}) \) before evaluating \( q \) on \( \tilde{S} \). A simple but effective technique [3] for deciding whether \( q \) should be forwarded to \( \tilde{S} \) is based on computing an upper-bound \( m(q, \tilde{S}) \) for \( s(q, 1, \tilde{S}) \) and comparing this bound against \( s(q, k, \tilde{S}) \). If \( m(q, \tilde{S}) \leq s(q, k, \tilde{S}) \) holds, it is guaranteed that \( \tilde{S} \) has no better documents than those in \( \tilde{S} \) and there is no need to forward \( q \) to \( \tilde{S} \). Otherwise, \( \tilde{S} \) may have better documents, and \( q \) has to be forwarded to \( \tilde{S} \). As the gap between \( m(q, \tilde{S}) \) and \( s(q, 1, \tilde{S}) \) increases, the query is more likely to be forwarded to a site with no useful documents. Hence, the objective in this thresholding technique is to compute \( m(q, \tilde{S}) \) value as tight as possible, i.e., this bound should be as close as possible\(^7\) to \( s(q, 1, \tilde{S}) \).

#### 3.2 LP Formulation

Assume that we have the precomputed \( s(q_i^q, 1, \tilde{S}) \) value for every query \( q_i^q \) in a set \( Q' = \{q_1^q, \ldots, q_m^q\} \) of \( m \) offline queries. Each \( q_i^q = \{t_1^q, \ldots, t_n^q\} \) is composed of unique terms. We are given an online query \( q = \{t_1, \ldots, t_n\} \) at local site \( \tilde{S} \) with a \( k \)th local score \( s(q, k, \tilde{S}) \). Given these, we formulate the problem of computing a tight \( m(q, \tilde{S}) \) value as a linear programming (LP) problem as follows. We first introduce a real-valued variable \( x_{t_j^q} \) for each term \( t_j^q \). We then find every offline query \( q' \in Q' \) such that \( q' \) is a proper subset of \( q \), i.e., \( q' \subset q \). For every such \( q' \), we introduce an inequality

\[
\sum_{t_j^q \in q'} x_{t_j^q} \leq s(q', 1, \tilde{S}), \quad \forall q', \text{ s.t. } q' \in Q' \text{ and } q' \subset q,
\]

which always holds. We also have the set of inequalities

\[
x_{t_j^q} \geq 0, \quad \forall t_j^q, \text{ s.t. } t_j^q \in q,
\]

which guarantee that the top scores for single-term queries (i.e., query terms) are always non-negative. After this setting, the thresholding problem reduces to finding

\[
m(q, \tilde{S}) = \max_{t_j^q \in q'} \sum x_{t_j^q}
\]

subject to the linear constraints given in Eqs. 7 and 8 via linear programming. In practice, there exist well-known, efficient LP solvers for this and similar problems.

\(^5\)Refer to Section 6.1 in [3] for more background on scoring.

\(^6\)We omit superscripts on symbols for better readability.

\(^7\)However, the inequality \( m(q, \tilde{S}) \geq s(q, 1, \tilde{S}) \) always holds.
We now illustrate the formulation by an example. Let \( q = \{ t_1, t_2, t_3, t_4 \} \) and \( Q' = \{ q_1', q_2', \ldots, q_9' \} \) with \( q_1' = \{ t_1 \} \),
\( q_2' = \{ t_2 \} \), \( q_3' = \{ t_3 \} \), \( q_4' = \{ t_4 \} \), \( q_5' = \{ t_1, t_2 \} \), \( q_6' = \{ t_2, t_3 \} \), and \( q_9' = \{ t_2, t_4, t_3 \} \). Assume that precomputed top scores are \( s(q_1', \tilde{S}) = 9.7 \), \( s(q_2', \tilde{S}) = 8.1 \), \( s(q_3', \tilde{S}) = 3.2 \), \( s(q_4', 1, \tilde{S}) = 4.9 \), \( s(q_5', 1, \tilde{S}) = 4.2 \), \( s(q_6', 1, \tilde{S}) = 4.7 \), and \( s(q_9', 1, \tilde{S}) = 5.1 \). These lead to the following set of inequalities
\[
\begin{align*}
x_1 & \leq 9.7; \quad x_2 \leq 8.1; \quad x_3 \leq 3.2; \quad x_4 \leq 4.9; \\
x_1 + x_2 & \leq 4.2; \quad x_2 + x_3 \leq 4.7; \quad x_4 + x_4 \leq 5.1; \\
x_1 & \geq 0; \quad x_2 \geq 0; \quad x_3 \geq 0; \quad x_4 \geq 0.
\end{align*}
\]
The maximum score satisfying these constraints is found as \( m(q, \tilde{S}) = 9.3 \) (\( x_1 = 4.2, x_2 = 0, x_3 = 0.2, x_4 = 4.9 \)).

### 3.3 Query Forwarding Algorithm

The forwarding algorithm contains an offline and an online phase. In the offline phase, offline queries are generated first. This can be done in different ways (see Section 4.4), e.g., synthetically by combining popular terms in the document collection or by extracting popular queries in past user query logs. Assuming such a set is available, the top scores are then computed for every query in this set over all local indexes. The computed values are replicated on all sites.

In the online phase, a query \( q \) is processed as follows. First, we evaluate \( q \) locally on \( S \) and record \( s(q, k, \tilde{S}) \). If a query term \( t_j \in q \) does not appear in any of the offline queries, \( q \) is forwarded to every non-local site (case \textsc{F-MissingInfo}) as it is impossible to compute any score bounds. For a non-local site \( \tilde{S} \), if there is a subquery \( q' \subseteq q \) for which \( m(q, \tilde{S}) = 0 \), \( q \) is not forwarded to \( \tilde{S} \) (case \textsc{L-ZeroThreshold}). Finally, for each non-local site \( \tilde{S} \) for which no decision is yet made, we separately solve the LP formulation of the previous section.

If \( m(q, \tilde{S}) > s(q, k, \tilde{S}) \) holds, \( q \) is forwarded to \( \tilde{S} \) (case \textsc{F-HighLPBound}); otherwise, it is not (case \textsc{L-LowLPBound}). We note that, in our LP formulation, it is possible to capture the \textsc{F-MissingInfo} case simply by introducing an equation \( x_j \leq \infty \) for every term \( t_j \in q \) such that \( t_j \notin q' \) for \( \forall q' \in Q' \).

### 4. EXPERIMENTAL FRAMEWORK

#### 4.1 Setup

We simulate a geographically distributed search engine architecture using two different setups. The first setup, referred to as Europe, consists of five search sites, located in Berlin (Germany), Madrid (Spain), Paris (France), Rome (Italy), and London (UK). The second setup, referred to as World, includes five relatively distant search sites, located in Canberra (Australia), Brazil (Brazil), Ottawa (Canada), Berlin (Germany), and Mexico City (Mexico). These two setups represent search architectures where the network latencies between the sites are low and high, respectively.

For both setups, we approximate the site-to-site network latency between any two sites by taking into account the speed of light on copper wire (200,000 km/s) and the bird-flight distances between the cities that the sites are located. To approximate the user-to-site latency between a site and its users, we take an average over the latencies between the capital city where the site is located and the most populated five cities in the respective country. Computing latencies this way is reasonable as latencies are known to correlate well with geographical distance [12], and our data transfer costs are negligible as transferred result sets are very small. However, this approach ignores queuing delays and the fact that network connections are not necessarily on straight lines. Therefore, using several, geographically distant computers, we measured real network latencies and obtained a mapping from predicted latencies to actual values through regression (Fig. 2). All predicted values are converted to final, more accurate latency values via this mapping. Table 1 displays some statistics about the latency values used in our setups.

#### 4.2 Dataset

As the global document collection, we use a large crawl of the Web (about 200 million documents). This collection is obtained through various cleansing and filtering steps. Hence, it is high-quality and its documents have high potential to appear in real-life search results. Then, using a proprietary classifier\(^9\), a home country is predicted for every document, and disjoint subsets of documents are assigned to search sites (some documents are not assigned to any site). Finally, separate indexes are built on each subcollection.

For each site, we extract consecutive queries (about 19 million queries in total) from the query logs of the Yahoo! web search engine. Queries are passed through cleansing steps, such as case-folding, stop-word elimination, term unifying, and reordering of query terms in alphabetical order. We omit queries issued by clicking on the next link and use only first page requests\(^10\). The query set of each site is separately sorted in increasing order of arrival times. The last quarter of each query set is used in the online phase. The rest are used in the offline phase. In our sample log, most queries are regional and occur in one site (Fig. 3).

#### 4.3 Simulations

We compute thresholds and local top \( k \) results using a modified version of Terrier. In simulations, we assume that each search cluster node builds an index on three million documents. The total number of processors available to the overall search system is determined accordingly. Each site is assigned a number of processors proportional to its index size. Therefore, query processing times are comparable

---

\(^9\)This is a production-level classifier that uses features such as language, IP, and domain to identify documents’ regions.

\(^10\)Next page requests may be handled by prefetching of result pages [14]. This is beyond the scope of this paper.
for search sites\textsuperscript{11}. Query evaluation is simulated via a detailed simulator, which computes a separate response time for each query, using Eqs. (3) and (4). In query processing, we assume a processing cost of 200ns per posting. This is an average value obtained from Terrier, but we observed it to correlate well with real search engine timings. We also assume a 20ms overhead for preprocessing, per query.

### 4.4 Offline Query Generation

Our LP-based solution is applicable to online queries of arbitrary length and is especially suitable for long queries. However, our query set is composed of web queries, which are very short in nature. Using long queries in the offline query set does not bring much additional performance benefit\textsuperscript{12}. Therefore, in our offline query set, we consider only single- and two-term queries\textsuperscript{13}. This approach also reduces the storage requirement for the precomputed scores and their offline computation cost. Moreover, if we assume that an offline query can be accessed in $O(1)$-time using a hash table, the computational cost of accessing offline queries that are proper subsets of the online query becomes much lower as this can be done in $O(q^2)$-time instead of $O(2^q)$-time.

In this work, we generate two offline query sets, referred to as $D_1$ and $D_2$. $D_1$ contains all terms (i.e., queries of length one) in the vocabulary of the collection. $D_2$ contains all possible pairs of vocabulary terms (i.e., queries of length two). Obviously, the latter may not be feasible in a practical setting, but we still prefer to experiment with this set to observe the degree of benefit that our thresholding algorithm may provide. We also generate two more query sets, $Q_1$ and $Q_2$, using the query log. $Q_1$ contains, as an offline query, all the terms in the vocabulary of the query log. $Q_2$ contains subqueries of length two in each individual query in the log (but, not across the entire vocabulary as we do for $D_2$).

For performance evaluation, we use selected combinations (unions) of the above-mentioned sets: $Q_1$, $D_1$, $Q_1$-$Q_2$, $D_1$-$Q_2$, and $D_1$-$D_2$. We omit combinations $Q_2$, $D_2$, $Q_1$-$D_1$, and $Q_1$-$D_2$ as they are less meaningful or useful. In our experiments, we use the $D_1$ set as our baseline as this is identical to the technique discussed in [3] (see the discussion in Section 7). To measure the best possible performance, we also consider an Oracle algorithm that has no false positives, i.e., it forwards a query to only the non-local sites with positive contribution to the final result set. Due to space limitations, occasionally, we display the results for only a single setup (often, Europe).

### 5. PERFORMANCE

#### 5.1 Effect of Offline Query Set

Fig. 4 shows the fraction of locally processed queries for different offline query sets, as $k$ varies. It is interesting to observe that the $Q_1$-$Q_2$ set outperforms the baseline ($D_1$) although it has fewer queries. When the baseline is combined with the term pairs extracted from the query log (i.e., $D_1$-$Q_2$), for $k=10$, about 9.1% more queries are processed locally (10.2% for the World setup). The impractical $D_1$-$D_2$ set performs quite close to the Oracle algorithm, which processes about 40% of the queries locally, for $k=10$.

\textsuperscript{11}We assume that indexes are entirely kept in main memory.

\textsuperscript{12}A similar issue is mentioned before in the context of caching intersections of posting lists [15].

\textsuperscript{13}We implemented an LP solver, tailored to our purposes.

The increase in the fraction of locally processed queries leads to a reduction in average query response times (Fig. 5). However, the distribution of response times is also important, i.e., we should check what fraction of queries can be processed under a given threshold time. It is empirically shown that after a certain response time threshold, users become frustrated and URL click-through rates go down, leading to financial losses for the search engine [16]. It is also shown that, given additional time for query processing, it is possible to improve the quality of search results [8]. In our simulations, we observe that the response time is a more critical issue for the World setup than Europe (Fig. 6). For Europe, only less than 10% of the queries cannot be answered under 400ms, whereas this rate varies in the 40%–55% range for the World setup, depending on the offline query set used.

According to Fig. 7, our technique achieves considerable reduction in the average number of non-local sites contacted. Additionally, in Fig. 8, we show the average number of sites that are active in processing a query. The second excludes any site that does not process the query on its local index. This may happen due to absence of a query term in the site’s index, which explains the overlap of curves in Fig. 8 for $Q_1$ and $D_1$ as well as $Q_1$-$Q_2$ and $D_1$-$Q_2$. Fig. 9 shows the average relative workload values as computed by Eq. (6). We observe that, for $k=10$, there is about 16% reduction in the workload when queries are evaluated over our architecture (assuming $D_1$-$Q_2$) relative to query evaluation over the full index (this goes up to 20% for the World setup).

Fig. 10 shows the average outcome of a forwarding decision for a (query, site) pair (recall the discussion in Section 3.3). In the figure, we observe the following: Since $Q_1$ and $Q_1$-$Q_2$ sets miss some terms in the collection vocabulary, about 10% of test queries had to be forwarded to all non-local sites (case F-MissingInfo). Most of the improvement
over the baseline is due to the proposed LP solution (cases F-HighLPBound and L-LowLPBound), which uses term pairs (e.g., see D1 versus D1-Q2). The fraction of L-ZeroThreshold cases correlates with the size of the offline query set used.

5.2 Effect of Partial Index Replication

Replicating globally popular documents on all sites leads to a high reduction in the forwarded query count [3]. The algorithm in [3] (herein, we call it R-freq) sorts the documents globally in decreasing number of occurrences in the top 200 results of training queries. A certain fraction of the most frequent documents are then replicated and indexed on all sites. This type of replication has two benefits for thresholding algorithms: local kth scores get higher as local sites have more documents, and thresholds computed for offline queries on non-local sites get lower as they are now computed over fewer documents. Both imply less forwarding.

A possible improvement over R-freq is to incorporate the storage cost incurred on the index due to replication of the document. This is a greedy algorithm (R-cost) that tries to optimize per-byte benefit at any step by prioritizing documents according to the ratio of their occurrence frequencies and storage costs. In related experiments, we compute the occurrence frequencies using the top 10 results of training queries and then replicate on all sites 0.5% of documents with the highest benefit. According to Figs. 11 and 12, surprisingly, the improvement achieved by R-cost over R-freq is minor (0.3%–0.9% increase in the rate of locally processed queries and 0.9%–2.8% decrease in the number of non-local sites contacted per query). This is mainly because R-cost fills the given replication budget with small documents that have low past occurrence frequencies. Although past occurrences correlate well with future occurrences at high frequency values, there is little correlation at low occurrence frequencies. This limits the performance of R-cost.

5.3 Effect of Result Caching and TTL

Search engines cache the results of frequent and/or recent queries to reduce the query workload on backend clusters and improve query response times [2]. Queries that result in a hit in the cache are served by the cache. In our context, result caching has a significant impact on the number of forwarded queries. With result caching, the fraction of queries that can be locally processed increases by 35%–45%, depending on the offline query set used (Fig. 13). We also observe that more informative query sets receive a lower benefit. This is because, under result caching, only the queries that are seen for the first time (i.e., compulsory cache misses) are subject to forwarding. Most cache misses are tail queries, which are long. As we will see in Section 6.3, long queries are much less likely to be forwarded, and hence having more information in bound computations becomes less important.

The above discussion holds for search engines with indexes that are periodically rebuilt (e.g., once a week). If, however, there are incremental updates on the index, result cache entries may become stale. In practice, a quick solution is to associate a fixed time-to-live (TTL) value t with every cache entry [7]. A cache entry that is not refreshed for at
6. FURTHER OPTIMIZATIONS

We now describe various techniques that improve performance under certain conditions. Some of the optimizations in distributed IR are also applicable to our setting, e.g., non-local computations can be early terminated simply by transferring \( s(q, k, \hat{S}) \) values together with the query. However, here, we skip such techniques and focus on those that are more meaningful in a geographically distributed setting.

6.1 Non-local Top k Optimization

If a query is forwarded to a non-local site, the top \( k \) results are requested. However, we note that it suffices to request \( k-r \) results, where \( r \) is the lowest rank such that \( m(q, S) < s(q, r, \hat{S}) \) holds. Hence, for some queries, it is possible to request fewer documents and reduce the number of remotely computed snippets, in addition to other savings in score computations. For \( k = 10 \), the saving is in the 5.3\%–16.7\% range, depending on the offline query set (Fig. 15).

6.2 Early Result Presentation

Search results are typically displayed in pages, containing 10 results. An interesting optimization is to show the user the local site’s search results without waiting for replies of non-local sites. If it later turns out that \( s(q, k, \hat{S}) \geq s(q, 1, \hat{S}_i) \) for every non-local site \( \hat{S}_i \), the query becomes served at the speed of a local query. Otherwise, non-local results are merged as a background job and the user’s screen is refreshed with the correct results\(^{17}\). In our case, for about a quarter of queries, all top 10 results come from the local site. For the top result, this is so for more than half of queries (Fig. 16).

6.3 Early Query Forwarding

We note that if \( m(q, \hat{S}) > \sum_{t \in S} s(t, \lfloor (k-1)/|q| \rfloor + 1, \hat{S}) \) holds\(^{18}\) \( q \) can be immediately forwarded to \( \hat{S} \) without waiting for completion of the local evaluation, which determines \( s(q,k,\hat{S}) \). This way, it becomes possible to overlap local query evaluation with network transfer. This approach requires precomputing and storing \( s(t, \lfloor (k-1)/|q| \rfloor + 1, \hat{S}) \) values for all terms in the collection vocabulary. Since the stored value depends on query length, covering all query lengths (assuming \( k \) is fixed to 10) requires storing all \( s(t,r,\hat{S}) \) values for \( r \in \{1, 2, 3, 4, 5, k\} \). If long queries (according to Table 2, less than 15% of queries have more than 3 terms) are ignored, it suffices to store the scores only for \( r \in \{4, 5, k\} \).

Herein, we only store \( s(t,k,\hat{S}) \) values and observe the impact on queries with a single term (about one-fifth of forwarded queries have one term, as seen in Fig. 17). For affected queries, the response time saving is up to 18ms (Table 3).

6.4 Remote Result Preparation

If the query is forwarded to only a single non-local site, the final results can be created and returned to the user by the non-local site as there are only two sets to be merged. This approach requires transferring the local top \( k \) result set together with the query, but may reduce the overall network latency in returning results to the user. Table 4 shows that there is a correlation between the average query response time and the number of non-local sites a query is forwarded to. The gap between the response time of local and forwarded queries is clearly seen. Our optimization, however, is applicable to a limited set of queries. According

\(^{16}\)We use small TTL values that are suitable to our sample query set. In practice, the TTL values are around a day [7].

\(^{17}\)Some vertical search sites use similar optimizations (e.g., http://www.kayak.com). The impact of this kind of result presentation on user satisfaction is open to investigation.

\(^{18}\)The right-hand side is an upper bound on \( s(q,k,\hat{S}) \).
to Fig. 18, about only 10% of our queries are forwarded to a single non-local site. For such queries, the saving in query response time is between 10ms and 16ms (Table 5).

7. RELATED WORK

Although there is much research on distributed IR [1], little research is done on multi-site, distributed search engines [3, 8]. Cambazoglu et al. present cost models and results, showing the potential of multi-site architectures for efficiency and relevance improvements [8]. Baeza-Yates et al. develop analytical models to compare operational costs of multi-site search systems against centralized systems [3].

The work in [3] is the closest to ours in that it also proposes an algorithm that tries to increase the number of locally processed queries by a thresholding technique, based on precomputation of maximum score contributions for all terms in the global vocabulary and replicating this information on all search sites. This way, it becomes possible to locally compute the maximum score a document can get on a non-local site, simply summing the maximum possible scores for query terms without evaluating the query remotely at all. It turns out that this technique is a limited case of our general solution (the same as using D1 in our setting).

We note that the query forwarding problem we deal with is somewhat different than the collection selection problem in federated IR [6]. In our system, queries are evaluated over a local index and some of them are forwarded. In federated IR, all queries are forwarded without any evaluation on a local index. We further note that P2P search systems [17] are also very different due to existence of a very high number of peers, their volatile nature, and limited availability. Baeza-Yates et al. [5] describe an architecture in which the global index is split into two tiers. In this architecture, queries are evaluated on one or two tiers, based on the decision of a machine-learned corpus predictor. In that architecture, documents are split into tiers by their importance or location.

Finally, Das et al. employ an LP-based solution in the context of databases for top k computation using materialized views [10]. Kumar et al. apply a similar technique to generalize top k thresholding algorithms by using precomputed intersections of posting lists [13]. To our knowledge, there is no work applying a similar technique in our context.

8. CONCLUSIONS

We showed that the fraction of locally processed queries in a multi-site search engine can be significantly increased by using an LP-based thresholding technique, and results are further improved by caching and replication. There are three research directions surfaced by our work. First, the applicability of our techniques to other problems (e.g., tiering [5]) should be investigated. Second, a trade-off analysis is needed between forwarding performance and offline query generation and storage overheads. Finally, the freshness of precomputed score thresholds needs further research.

9. ACKNOWLEDGEMENT

This publication is based on work performed in the framework of the Project COAST-ICT-248036, funded by the European Community, and partially supported by the Scientific and Technological Research Council of Turkey under grant EEEAG-109E019 and COST Action IC080 ComplexHPC.

10. REFERENCES

Query Processing in Replicated and Term-Partitioned Inverted Indexes
R. Oguz Selvitopi¹, Ata Turk¹, Cevdet Aykanat¹, B. Barla Cambazoglu¹

Abstract

Due to recent advances in search technologies, utilizing term-partitioned indexes in parallel query processing became a viable alternative. In term-partitioned inverted indexes the performance of the system is improved via replication of terms and the state-of-the-art approach is the replication of most-frequent terms. In this study, we adopt a recently proposed replication approach based on replicated hypergraph partitioning, which utilizes query logs. We also consider the problem of scheduling which occurs when replication is involved. We investigate these schemes in an actual parallel query processing system utilizing a term-partitioned inverted index where replication of terms are applied to improve performance. We provide extensive experimental analysis performed up to 32 processors to show that proposed schemes are far more superior to the state-of-the-art alternatives.

Keywords:

1. Introduction

In modern text retrieval, the main objective of query processing is to find out the relevant documents to a user query and displaying them to the user. The vector space model [1] is used to calculate the similarity between a user query and documents in the collection. A set of documents is returned to the user according to the result of these similarity calculations. This document set is sorted in decreasing order with respect to similarity to the user query. tf-idf together with the vector space model [2] is considered in this work.

Most common method for storing large document collections is using inverted indexes [3, 4]. In the inverted index data structure, there is an associated list of documents for each term. These lists of documents are also called postings/inverted lists. An inverted index contains an inverted list (also called posting list) for every term in the data collection. Each posting of an inverted list associated with a term consists of a document id field and a weight field for each document containing that term. The weight field is the result of a weight function [5] and shows the relevance between the document and term associated with that posting.

The growing use of the internet has a significant influence on text retrieval systems. The size of the text collection available online is growing at an astonishing rate. At the same time, the number of users and queries submitted to the text retrieval systems are increasing very rapidly. The staggering increase in the data volume and query processing load create new challenges for text retrieval research. In order to satisfy user needs when large volumes of data is being processed, usage of parallel methods becomes inevitable. Parallel frameworks provide better average response times and higher throughput rates compared to sequential methods.

In general a data-parallel approach is adopted for parallelizing the retrieval process on shared-nothing architectures, where the inverted index is distributed among multiple index servers. Then the query processing computations are distributed among the index servers according to the owner computes rule. The query responses are generated by combining the partial answer sets produced by the index servers.

In general, distribution of the inverted index can be performed in either document-based or term-based fashion. In document-based distribution, a set of documents in the dataset is assigned to a particular index server. During query processing, each index server contributes to the final answer set by computing the similarities of the documents assigned to itself. Hence, each query must be sent to all index servers. The answer sets produced by the index servers are merged to form the final answer set. In term-based distribution, each inverted list is assigned to an index server. For each query, a subquery should be sent to the index servers containing at least one term within the query. Only the index servers receiving a subquery is required to respond with a partial answer set in order to compute the final answer set.
In this distribution, the partial answer sets are not sufficient to decide whether a document is qualified to be in the final answer set or not. The results of all participant index servers should be accumulated since the terms of a document are scattered throughout separate index servers.

In state-of-the-art search engine systems, document-based partitioning is preferred due to the following reasons: (i) following a parallel crawling phase, building a document-based partitioned index becomes easier than building a term-based partitioned index, (ii) document-based partitioned indexes achieve better load-balancing performance during query processing. However, due to the recent advances in the search technology, the search engines keep their inverted indices in memory to reduce query response times and as shown in a recent work [6], building term-based partitioned inverted indexes in memory can be achieved in acceptable times. Furthermore, the load-balancing inferiority of term-based partitioned inverted indexes can be alleviated via term-replication schemes. Generally, these replication schemes replicate most frequently queried terms on all processors and apply dynamic load-balancing schemes to improve the performance. Thus, together with replication, term-based partitioning becomes a viable alternative to doc-based partitioning.

In this study, we consider a parallel query processing system utilizing a term-partitioned inverted index where replication of terms are applied to improve performance. Our contributions are fourfold:

- We implement a successful parallel query processing system and we provide details of our implementation and the reasons behind our design choices.

- We utilize a successful replication approach based on replicated hypergraph partitioning, which was recently proposed in [7]. Our experimental results demonstrate that this approach is much more successful than the state-of-the-art partitioning and replication schemes.

- When there is replication, the problem of selecting the replica to be used in query processing problem arises. We show that this problem can be reduced to the set-cover problem. Furthermore, we propose various heuristics for scheduling of replicated terms.

- We provide extensive experimental analysis performed up to 32 processors on our parallel query processing system.

The rest of the paper is organized as follows: In Section 2, we provide the related work and background. In Section 3, we describe the details of our parallel query processing system. In Section 4, we analyze various partitioning and replication schemes. In Section 5, we discuss scheduling issues in replicated term-partitioned indexes. Finally, in Section 6, we compare proposed methods and algorithms with state-of-the-art techniques and analyze and discuss the results.

2. Background and Related Work

2.1. Related Work

In this section, we discuss the most commonly used approaches for replication in parallel information systems. In [7], the authors analyze caching and partial replication to improve the performance of parallel information retrieval systems. In their approach, they replicate the documents requested by previous queries (thus, using a query log) to a distinct server, which is called the partial replica. This partial replica is further used to answer the future queries if possible. In their work, they show that the replication is able to reveal access locality of the queries. In [8], the performance of clustered and replicated IR systems are investigated. In their work, a document-based index distribution is assumed and a simulation environment is used to compare these two systems. In their approach, a replicated system consists of identical distributed systems, where each distributed system contains all collection and the brokers decide which replica will be used for a given query. On the other hand, a clustered system consists of disjoint sets of documents, where each cluster can be distributed or replicated. Their experiments show that the clustered system does not outperform the replicated system and the brokers and the network can be bottleneck in both of these systems. In Google cluster [9], which consists of hundreds of commodity machines, replication is mainly used for performance (i.e. throughput) and availability. The index is partitioned in a doc-based manner and replication is used in all levels of the cluster architecture including the hardware-based solutions (RAID), the cluster itself and across the clusters. The authors of [10] use the query log to obtain information about term frequencies which is further utilized for the distribution and replication of terms to index servers. Their term distribution approach is based on a heuristic that is generally used for the bin-packing problem. In their replication method, they replicate a certain amount of most frequent terms and their inverted lists to all index servers. Performing term distribution and replica-
tion as mentioned, they show that they can achieve better load balancing for term-based partitioned indexes. Another recent research utilizing query logs is [11]. As in [10], they first distribute the terms based on a bin-packing heuristic using index servers as bins and terms as items. For the remaining terms which do not appear in query log, they distribute them to the index servers so that each index server possesses approximately equal number of terms. They also investigate replication by replicating a certain percentage of most frequent terms to all index servers. Thus, by using query logs to obtain a better term distribution and replication, they are able to achieve improvements in response time and throughput.

2.2. Hypergraph Partitioning

A hypergraph \(H=(V, N)\) is defined as a set of vertices \(V\) and a set of nets \(N\). Each net \(n_j \in N\) connects a subset of vertices. The set of vertices connected by net \(n_j\) is denoted as \(Vertices(n_j)\). The set of nets that connect vertex \(v_i\) is denoted as \(Nets(v_i)\). The vertices \(v_i\) and \(v_j\) are said to be neighbors if they are connected by at least one common net, i.e., \(Nets(v_i) \cap Nets(v_j) \neq \emptyset\). An \((n_j, v_i)\) tuple denotes a pin of \(n_j\) where \(v_i \in Vertices(n_j)\). The degree of a net \(n_j\) is equal to the number of vertices it connects, \(|Vertices(n_j)|\). The total number of pins \(P = \sum_{n_j \in N} |Vertices(n_j)|\) denotes the size of a given hypergraph \(H\). A weight value \(w(v_i)\) is associated with each vertex \(v_i\) and a cost value \(c(n_j)\) is associated with each net \(n_j\). The cost function for a net easily extends to a subset of nets \(M \subseteq N\), i.e., \(c(M) = \sum_{n_j \in M} c(n_j)\).

\(\Pi=[V_1, \ldots, V_k]\) is a \(K\)-way partition of \(H=(V, N)\) if each part \(V_i\) is a nonempty subset of \(V\), parts are pairwise disjoint and union of \(K\) parts is equal to \(V\). The weight \(W(V_k)\) of a part \(V_k\) is the sum of the weights of the vertices in that part, i.e., \(W(V_k) = \sum_{v_i \in V_k} w(v_i)\). A partition \(\Pi\) is said to be balanced if each part \(V_k \in \Pi\) satisfies the balance constraint:

\[
W(V_k) \leq (1 + \epsilon)W_{\text{avg}} \quad \text{for} \quad k = 1, \ldots, K
\]  

(1)

where \(W_{\text{avg}} = W(V)/K\) and \(\epsilon\) is predetermined maximum imbalance ratio.

In a partition \(\Pi\), a net is said to connect a part if it connects at least one vertex in that part. Connectivity set \(\Lambda(n_j)\) of a net \(n_j\) is defined as the set of parts connected by \(n_j\). The number of parts in the connectivity set of \(n_j\) is denoted by \(\lambda(n_j)=|\Lambda(n_j)|\). A net is said to be cut or external if it connects more than one part \((\lambda(n_j)>1)\), and uncut or internal if it connects only one part \((\lambda(n_j)=1)\). The set of external nets in a partition \(\Pi\) is denoted as \(N_E\). The set of internal nets that connect a vertex \(v_i\) is denoted as \(\text{InternalNets}(v_i)\). Two cutsize metrics widely used in the literature to represent the cost of a partition \(\Pi\) are:

\[
cutsize(\Pi) = \sum_{n_j \in N_E} c(n_j).
\]  

(2)

\[
cutsize(\Pi) = \sum_{n_j \in N_E} (\lambda(n_j) - 1)c(n_j).
\]  

(3)

The cost definitions in Equation 2 and Equation 3 are called as the cut-net and the connectivity metric, respectively. For example, the cut-net and connectivity metrics model the minimization of the communication volume in parallel sparse matrix vector multiplication utilizing collective and point-to-point communication schemes, respectively [12, 13].

Given a hypergraph \(H=(V, N)\), the Hypergraph Partitioning can be defined as finding a \(K\)-way partition \(\Pi=[V_1, \ldots, V_k]\) that minimizes the cutsize (Equation 2 or 3) while maintaining the balance constraint (Equation 1). This problem is known to be NP-hard [14].

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)</td>
<td>Terms in the collection, the lexicon</td>
</tr>
<tr>
<td>(D)</td>
<td>Documents in the collection</td>
</tr>
<tr>
<td>(K)</td>
<td>Number of index servers</td>
</tr>
<tr>
<td>(IS_k)</td>
<td>(k^{th}) index server</td>
</tr>
<tr>
<td>(t_i)</td>
<td>A term in the collection</td>
</tr>
<tr>
<td>(\ell^t)</td>
<td>The instance of the replicated term (t_i) at (IS_k)</td>
</tr>
<tr>
<td>(d_j)</td>
<td>A document in the collection</td>
</tr>
<tr>
<td>(L)</td>
<td>The set of inverted lists</td>
</tr>
<tr>
<td>(\Lambda)</td>
<td>The set of inverted lists associated with (IS_k)</td>
</tr>
<tr>
<td>(IS)</td>
<td>The inverted list associated with (t_i)</td>
</tr>
<tr>
<td>(p_i)</td>
<td>A posting in an inverted list</td>
</tr>
<tr>
<td>((d_i, w_i))</td>
<td>The document and the weight associated with (p_i)</td>
</tr>
<tr>
<td>(q_i)</td>
<td>A user query</td>
</tr>
<tr>
<td>(\tilde{q}_i)</td>
<td>The subquery generated for (q_i)</td>
</tr>
<tr>
<td>(\tilde{p}_i)</td>
<td>The partial answer set for (q_i) constructed by (IS_k)</td>
</tr>
<tr>
<td>(AA)</td>
<td>Accumulator array</td>
</tr>
<tr>
<td>(a_j)</td>
<td>An entry in the accumulator</td>
</tr>
<tr>
<td>((d_i, s_i))</td>
<td>The document and the score associated with (a_j)</td>
</tr>
<tr>
<td>(O_r)</td>
<td>The queue of receptionist</td>
</tr>
<tr>
<td>(Q_k)</td>
<td>The queue of (IS_k)</td>
</tr>
<tr>
<td>(r)</td>
<td>Maximum partial answer set size for an index server</td>
</tr>
<tr>
<td>(s)</td>
<td>The number of documents to be returned to the user</td>
</tr>
<tr>
<td>(A_d)</td>
<td>MAY NOT BE NECESSARY (only used once?)</td>
</tr>
</tbody>
</table>

3. Parallel Query Processing

In this section we explain the details of our parallel query processing system, as well as some important design choices and the reasons behind these decisions. We
assume a shared-nothing parallel architecture with $K$ index servers and a single central receptionist, where the central receptionist and each index server are running on separate nodes. Interprocessor communication and coordination are achieved via explicit message passing. A PC cluster forms a typical case of our target architecture.

3.1. Basics of Query Processing

The main objective of query processing is to find out the relevant documents to a user query $q = \{t_1, t_2, \ldots, t_n\}$ and display them to the user. A set of documents $A_q = \{d_1, d_2, \ldots, d_s\}$ for $q$ is returned back to the user according to the result of various similarity calculations [1] between the documents in the collection and the query, where the found relevant documents are sorted in non-increasing order with respect to used similarity measure. In this work, we use the widely accepted tf-idf weighting scheme together with the vector space model [2] for similarity calculations. However, any other similarity measure can easily be integrated into our system.

Since it is not practical to make the similarity calculations directly using the raw document contents, documents are first converted into an inverted index [3, 4], $L = \{(t_1, I_1), (t_2, I_2), \ldots, (t_m, I_m)\}$. In the inverted index data structure, each term $t_i$ in the vocabulary of the collection has an associated inverted list $I_i$, which contains a set of postings $I_i = \{p_{i1}, p_{i2}, \ldots, p_{in}\}$. Each posting $p_{ij}$ of the inverted list associated with term $t_i$ is a tuple $(d_j, w_j)$, where $d_j$ is the id of a document that contains $t_i$ and $w_j$ [5] is the relevance between the document $d_j$ and the term $t_i$.

There are several phases (possibly interleaving depending on the implementation) of query processing: creation, update, extraction, selection, and sorting [15]. After a query is submitted for processing, an accumulator is created to store similarity scores for the documents. An entry in an accumulator is noted with $a_j = (d_j, s_j)$, where $s_j$ is reserved for storing the final similarity score of $d_j$ with respect to used similarity measure. During update phase, inverted lists corresponding to the terms in the query are fetched from the disk and the entries in the accumulator are updated accordingly. The update of these entries changes with respect to used document matching logic, which can be AND (conjunctive mode) or OR (disjunctive mode). In AND logic, only the documents that include all query terms are matched, whereas in OR logic, the documents that include at least one of the query terms are matched. The score $s_j$ for $a_j$ is simply computed by adding $w_j$ values of the corresponding postings for $d_j$ in the inverted lists of the terms of the query:

$$s_j = \sum_{t_i \in \delta I_j \land \delta I_j \in \delta t_j} w_j.$$

Then, the nonzero entries in the accumulator are extracted and the documents with the top $s$ scores are selected. Finally, the selected documents are sorted in non-increasing order of their similarity scores ($s_j$) and returned back to the user in this order. The reader is referred to [15] for an extensive analysis of alternative sequential query processing implementations.

3.2. Parallel Query Processing

3.2.1. Term-Based Index Distribution

In term-based index distribution, the inverted index $L$ is partitioned into $K$ pair-wise disjoint subsets $L_1, \ldots, L_K$, where $K$ is the number of index servers in the parallel query processing system. Formally,

$$L = \bigcup_{k=1}^{K} L_k; \quad L_i \cap L_j = \emptyset \quad \text{for} \ 1 \leq i < j \leq K. \quad (4)$$

Each index server $IS_k$ is responsible for maintaining the sub-index $L_k$, where

$$L_k = \{(t_i, I_j) : t_i \text{ is assigned to } IS_k\}.$$

The assignment of terms and their inverted lists to index servers can be performed in various ways. This process has a crucial effect on the performance of the parallel system and must be done carefully. We discuss this issue in great detail in Section 4.

A common technique used in improving performance of the parallel query processing systems is replication of terms and their inverted lists. Note that when replication of terms and their inverted lists are allowed, a term and its inverted list can be assigned to more than
For an example of a replicated term in Fig. 1, consider term \( t_i \) which has three replicas at \( IS_1, IS_2, \) and \( IS_4 \). To indicate the replica of \( t_i \) at \( IS_1 \), we use the notation \( t_i^1 \), for the replica at \( IS_2 \), it is \( t_i^2 \) and for \( IS_4 \), it is \( t_i^4 \).

### 3.2.2. Central Broker Scheme

The central broker parallel query processing scheme is a master-slave type of system. In a typical CB scheme, there is a single receptionist (master) which collects the incoming user queries and sends them to the index servers (slaves). The index servers are responsible for generating partial answer sets to the received queries using their local inverted indices, which are obtained via term-based partitioning (Section 3.2.1). The generated partial answer sets are later merged into a global answer set at the receptionist, forming the answer set for the query which is then sent back to the user. This section presents the proposed algorithms running on the receptionist and the index servers for the CB scheme. The implementation level details are given in the explanations of these algorithms.

Before starting to process user queries, the receptionist and the index servers allocate and initialize necessary data structures. Firstly, the receptionist creates a trie (also known as radix tree or prefix tree), in which it keeps the terms and their ids. Upon receiving a query, the id of a query term is accessed from the trie in \( O(\ell) \) memory accesses where \( \ell \) is the length of that term. A trie is preferred instead of a hash table because of its smaller memory consumption. The receptionist maintains a term-to-server map (simply referred as map) which contains information about the assignment of terms and their inverted lists to index servers in a term-based partitioning. The receptionist and the index servers use a queue while processing user queries. The queue maintained by the receptionist contains queries received from the users and the partial answer sets (PASs) received from the index servers, whereas the queue maintained by each index server contains only the subqueries received from the receptionist. Fig. 2 shows a snapshot of a parallel query processing system in central broker architecture that uses the term partitioning given in Fig. 1. A received query \( q \) with id \( i \) is denoted as \( q_i \). A subquery of \( q_i \) that is constructed for \( IS_k \) is denoted as \( q_i^k \). A PAS which belong to \( q_i \) and that was sent by \( IS_k \) to the receptionist is displayed by \( P^k_1 \). In the figure, the receptionist maintains three basic structures, a trie, a map and a queue \( Q \) which contains queries received from users as well as partial answer sets (denoted as \( P \) in the figure) received from index servers. An entry in the map at the receptionist indicates whether \( IS_k \) stores \( t_i \) and its inverted list. For example, \( t_4 \) and its inverted list are stored at two index servers \( IS_2 \) and \( IS_3 \) (see Fig. 1). In the map at Fig. 2, these corresponding entries are displayed as shaded. As mentioned, the queues at index servers \( (Q_1, Q_2, Q_3, Q_4) \) contain only subqueries received from receptionist.

**Index Server Algorithm.** As seen in Algorithm 1, our index server algorithm consists of a single infinite while loop (lines 1–18) in which the index server periodically probes for incoming subqueries, receives incoming subqueries, produces PASs for subqueries, and sends the produced PASs to the receptionist. As the first action of its loop, \( IS_k \) probes for incoming subqueries from the receptionist and if the probe value is true, receives the incoming subquery and enqueues it to its queue \( Q_k \) (lines 3–5). When a subquery is dequeued from the queue, \( IS_k \) processes the postings in the inverted lists of the subquery terms and updates the scores of documents in its accumulator array (AA). The update of the accumulator array (lines 9–11) changes with respect to used document matching logic. As an example consider
Algorithm 1: CB algorithm running on Index Server $IS_k$.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Input</strong> matchingLogic, $r$</td>
</tr>
<tr>
<td>2</td>
<td>while true do</td>
</tr>
<tr>
<td>3</td>
<td>PROBE whether a subquery is received from the receptionist</td>
</tr>
<tr>
<td>4</td>
<td>if PROBE = true then</td>
</tr>
<tr>
<td>5</td>
<td>Receive subquery $q_i^j$</td>
</tr>
<tr>
<td>6</td>
<td>if $Q_i \neq \emptyset$ then</td>
</tr>
<tr>
<td>7</td>
<td>$d_i^j$ ← DEQUEUE($Q_i$)</td>
</tr>
<tr>
<td>8</td>
<td>$AA \leftarrow \emptyset$ &gt; Initialize the accumulator array</td>
</tr>
<tr>
<td>9</td>
<td>foreach $t_j \in d_i^j$ do</td>
</tr>
<tr>
<td>10</td>
<td>foreach $(d_j, w_j) \in IS_j$ do</td>
</tr>
<tr>
<td>11</td>
<td>Update the $(d_j, t_j)$ entry in AA with respect to given matching logic</td>
</tr>
<tr>
<td>12</td>
<td>if matchingLogic = AND then</td>
</tr>
<tr>
<td>13</td>
<td>$P_i^j \leftarrow$ SELECT all nonzero entries in AA</td>
</tr>
<tr>
<td>14</td>
<td>else if matchingLogic = OR then</td>
</tr>
<tr>
<td>15</td>
<td>$P_i^j \leftarrow$ SELECT nonzero entries with top $r$ scores in AA</td>
</tr>
<tr>
<td>16</td>
<td>SORT $P_i^j$ with respect to document ids</td>
</tr>
<tr>
<td>17</td>
<td>WAIT for previous send to finish</td>
</tr>
<tr>
<td>18</td>
<td>SEND $P_i^j$ to the receptionist</td>
</tr>
</tbody>
</table>

For a query, the update of document scores begins at index servers and finishes at the receptionist since an entry $d_j = (d_j, s_j)$ may be existent in two or more PASs which belong to different index servers. In this case, the $s_j$ values need to be summed up at the receptionist to compute the final score of $d_j$. Due to this property, the index servers sort the entries of the PASs with respect to their document ids. This design choice is made so as to reduce the bottleneck at the receptionist by reducing its merge-and-update cost, at the expense of increasing the load at the index servers. Long accumulator arrays sent from the index servers to the receptionist induce high communication volume as well as high computational cost due to the merge-and-update operations at the receptionist. To alleviate this problem for OR document matching logic, we adopt a slight modification of the accumulator limiting approaches mentioned in [16], where we restrict our PAS size to $r$, which is set to a certain percent of the document collection size. As mentioned in the literature, this restriction does not degrade the query processing quality. Note that for AND document matching logic, the size limitation of PASs can have a great negative impact on the correctness and the quality of the returned answers, thus it is not possible to limit the size of PASs in AND logic. The reasons behind this phenomenon are explained later in this section in receptionist’s algorithm.

Returning to Algorithm 1, if document matching logic is AND (lines 12–13), all non-zero entries in the accumulator array are selected for the PAS $P_i^j$ (the PAS of $q_i$ formed at $IS_k$). If document matching logic is OR (lines 14–15), the nonzero entries with top $r$ scores are selected for $P_i^j$ in linear time. Then the index server sorts $P_i^j$ (line 16) with respect to document id fields of the entries, enabling the receptionist to merge-and-update PASs efficiently. Finally, the index server waits for previous send operations to finish and then sends $P_i^j$ to the receptionist (lines 17–18).

Receptionist Algorithm. We present the algorithm running on the receptionist in Algorithm 2. The receptionist firstly sets the variable globalId to 0 (line 1) which is used for numbering received queries. Then, the receptionist enters into an infinite while loop (lines 2–35), where it continuously checks if any queries or PASs are received, dispatches subqueries, merges and updates PASs and displays the final answer sets to users. As the first action within this while loop, the receptionist checks for incoming queries from users, and if any queries have been received, it enqueues them to its queue $Q_r$ (lines 3–6). Similarly, it checks for incoming PASs from each index server, and if any PASs have been received, it enqueues them to the same queue (lines 7–10).

The receptionist dequeues an item from its queue (line 12) and takes different actions depending on the type of the item dequeued. If the dequeued item is a query (lines 13–25), the receptionist first parses the query terms in order to find their ids using the trie and assigns an id to the query (lines 14–15). It then forms subqueries and distributes them to the index servers (lines 16–24). The subquery forming process includes another procedure called SCHEDULE, which basically, given a query and a map, returns information about the query terms on which index servers they will be processed. As an example, consider $q_5 = [t_1, t_3, t_7]$ whose
subqueries are already formed and distributed in Fig. 2. The call to SCHEDULE(q5, map) in this case returns \(\{IS_2, IS_3, IS_4\}\) which is assigned to variable qmap. This means \(t_1\) will be processed on \(IS_4\), \(t_2\) will be processed on \(IS_3\), and \(t_7\) will be processed on \(IS_4\). With this information, the subqueries are formed as \(q^5_1 = \emptyset\), \(q^5_2 = \emptyset\), \(q^5_3 = \{t_5\}\), and \(q^5_4 = \{t_1, t_7\}\). The non-empty subqueries \(q^5_3\) and \(q^5_4\) are then distributed to the index servers \(IS_3\) and \(IS_4\), respectively. Note that since replication is involved in the term-based partitioning, there are possibly multiple ways to form subqueries. In the example of scheduling terms of \(q_5\), all query terms \(t_1, t_3, t_5, t_7\) could be scheduled to \(IS_4\), or in an alternative scheduling \(t_1\) could be scheduled to \(IS_4\), \(t_3\) could be scheduled to \(IS_3\), and \(t_7\) could be scheduled to \(IS_2\). We investigate different scheduling heuristics in Section 5. Finally, the globalQid is incremented to be used in numbering of the next query (line 25).

If the dequeued item is a PAS (lines 26–35), the receptionist first retrieves the id of the query which PAS belongs to (line 27). We adopt a two-way merge-and-update algorithm for forming an answer set from received PASs for \(q_i\). In this approach, if the received PAS is the first one for \(q_i\), then it becomes the initial accumulator array (AA) for that query. Otherwise, the received PAS is merged-and-updated with the existing accumulator array immediately after it has been received. Clearly, before the last two-way merge, not all the scores are complete and the extraction and selection operations cannot be initiated.

The two-way merge-and-update algorithm can be considered as an extension of the conventional merge operation used in mergesort [9], which merges two sorted sub-lists into a sorted list in linear time. Like the merge operation, the algorithm advances two pointers over two document-id sorted accumulator arrays. In both AND and OR logic, if the two pointed entries’ document ids are the same, their scores are added and stored as a single entry in the resultant array. If they are not the same, in AND logic, since only common documents must be matched, the entry with the small document id is discarded by setting its score value to 0. However in OR logic, since there is no such restriction, the score value of the entry with smaller document id is directly stored in the resultant array.

For example, consider the query \(q_4 = \{t_1, t_4, t_7\}\) whose PASs are already in the queue of the receptionist in Fig. 2. Assume \(t_1\) is scheduled to be processed at \(IS_2\), \(t_4\) is scheduled to be processed at \(IS_3\), and \(t_7\) is scheduled to be processed at \(IS_4\). Given this information, the PAS contents regarding \(q_4\) in \(Q_4\) is \(P^4_4\), thus the accumulator array at the receptionist will be initialized to \([d_3, d_7, d_8]\). The next \(PAS\) to be processed for \(q_4\) in \(Q_4\) is \(P^4_3\), and the last one is \(P^4_2\). For AND logic, only the common documents will form the next accumulator array, thus after processing \(P^4_3\) the accumulator array will include \([d_4, d_7]\), and after processing \(P^4_2\) the accumulator array will only include \([d_4]\). For OR logic, union of the PASs will form the next accumulator array, thus after processing \(P^4_3\) the accumulator array will include \([d_4, d_7, d_8]\), and after processing \(P^4_2\) the accumulator array will include \([d_1, d_3, d_4, d_7, d_8]\). Note that in the update of common documents, the score values of these documents are summed (they are omitted for clarity). This example clearly illustrates why index servers need to send all of their computed PASs in AND logic: the receptionist is not only responsible for adding scores of documents (as in OR logic), but also selecting only the common documents that appear in all PASs for \(q_i\) for the final answer set.

The algorithm proceeds with merging-and-updating the dequeued PAS with the accumulator array with re-
spect to document matching logic (lines 28–31). If this PAS is the last PAS for \( q_i \), it means the answer set is ready to be sent back to the user. In this case, the receptionist selects the documents with top \( s \) scores in the accumulator array, sorts these documents with respect to score values, and then displays them to the user (lines 32–35).

4. Index Partitioning and Term Replication

Term-based index distribution and replication of terms and their inverted lists are explained in Section 3. In this section, we explain commonly used methods for achieving partitioning and replication in parallel query processing systems. Moreover, we describe our hypergraph-partitioning-based methods for achieving partitioning and replication.

4.1. Bin-packing-based Index Partitioning

In the bin-packing-based index partitioning scheme, each term is associated with a certain weight such as the frequency of the term in the query log or its inverted list length and assigning a term to an index server corresponds to assigning the load associated with that terms to that index server. Thus, obtaining a balance on the cumulative weights of the terms assigned to index servers corresponds to balancing the loads of these index servers. Obtaining a term-partitioned inverted index while minimizing the load (e.g. storage, computational, etc.) imbalance among the index servers can be reduced to the minimum makespan scheduling problem [17] if the index servers are identical. In this reduction, the terms correspond to tasks, the weights of terms correspond to task lengths, and index servers correspond to processors. In this way, minimizing the makespan corresponds to minimizing the load imbalance in term-partitioned indexes. Note that a different variation of minimum makespan scheduling problem with different processor speeds also exists [18].

Minimum makespan scheduling problem is known to be NP-hard [19], and thus, is generally addressed with heuristics. In parallel information retrieval systems, a best-fit decreasing heuristic that is generally used in bin-packing problem (hence the name) is reported to give good load balance values [10, 11] for obtaining term-partitioned indexes. This heuristic has the approximation value of 3/2 when applied to minimum makespan scheduling problem [17].

In the bin-packing heuristic for term-partitioned indexes, the weight associated with each term is generally (i) the inverted list length of that term, or (ii) the number of queries containing this term if the query log is utilized [10, 11]. If inverted list lengths are used, then this is equivalent to balancing the storage load of the index servers, which has an indirect and obscure effect on balancing computation and communication load. However, if query term frequencies are used, we actually try to balance the number of term accesses performed in each index server. Utilizing query logs is expected to give better computational load balance. Note that if query term frequencies are used, there may remain several terms that need to be assigned to index servers which do not occur in the query log. To distribute such unqueried terms, it is possible to follow the same bin-packing scheme given in [11] by using inverted list lengths of terms.

4.2. Most Frequent Term Replication

A common and widely used approach for replication in parallel query processing systems is the replication of most frequent terms [10, 11]. In this scheme, a certain amount of most frequent terms (with their inverted lists) are replicated in each index server. As mentioned in Section 4.1, the frequency of a term can be interpreted as either its inverted list length or the number of queries including this term if query log is utilized. Generally, the terms having longer inverted lists or more query term appearances have higher priorities for replication. Note that the most frequent term replication is independent of the underlying partitioning scheme.

The motivation behind the most frequent term replication is that the replication of high frequency terms, which are the most probable causes of load imbalance, is likely to improve the overall performance of the system in terms of average response time and query throughput. This is because by replicating such terms, we prevent bottlenecks in the system via distributing the loads of the highly accessed terms more evenly among the index servers. Thus, the most frequent term replication improves the performance of the parallel query processing system via using replication as a load balancing tool.

4.3. Hypergraph-partitioning-based Index Distribution

If query log is available, a hypergraph can be constructed utilizing the information in the log. Then, this hypergraph can be partitioned for the given number of index servers to obtain an index distribution. Given a query log consisting of \( m \) queries \( q_1, \ldots, q_m \), and the terms that appear in each query \( q_j = \{t_1, \ldots, t_r\} \) for \( j = 1, \ldots, m \), a hypergraph \( \mathcal{H} = (\mathcal{V}, \mathcal{N}) \) is constructed where the queries correspond to the nets and the terms correspond to the vertices of the hypergraph. More
specifically, a query $q_j = \{t_1, \ldots, t_r\}$ in the query log is modeled by a net $n_j$ and the terms of this query are modeled by vertices $v_1, \ldots, v_r$, where the vertices connected by $n_j$ correspond to terms that occur in $q_j$. Thus, if we have $m$ queries and a total of $n$ distinct terms that appear in the query log, the corresponding hypergraph model will consist of $m$ nets and $n$ vertices. After partitioning the constructed hypergraph $H$ and obtaining a partition $\Pi = \{V_1, \ldots, V_k\}$, the part $V_k$ is mapped to the index server $IS_k$ and each vertex $v_i \in V_k$ associated with $t_i$ and its inverted list is stored in $IS_k$. From now on when presenting hypergraphs in figures, we use $t_i$ and $q_j$ for vertices and nets, respectively.

Fig. 3 shows a four-way partition obtained using hypergraph partitioning for four queries. After obtaining a four-way partition, the terms and their inverted lists in obtained parts are assigned to the corresponding index servers which are illustrated in Fig. 3 as $IS_1, IS_2, IS_3$, and $IS_4$. The connectivity set of a net $q_j$ indicates which index servers will participate in answering this query. For example, the connectivity set of $q_3$ is $\{IS_2, IS_3, IS_4\}$. Thus, these index servers will participate in answering $q_3$.

In the hypergraph model we utilize, the weight of a vertex associated with $t_i$ is assigned the number of queries that include $t_i$ in the query log. In other words, we use query term frequencies as vertex weights. Therefore, maintaining balance in hypergraph partitioning corresponds to balancing the number of term and inverted list accesses (this is equivalent to the number of disk accesses) performed by each index server.

We assign unit costs to nets and use connectivity metric for cuto size computation to correctly capture the number of index servers involved in answering a query. In this model, the cuto size of a partition $\Pi$ corresponds to the total number of index servers involved in answering the queries in the log. Thus, we try to minimize the total number of index servers that are involved in answering queries which is especially useful for AND logic for document matching. That is because if a query can be answered from a single index server in AND logic, the inverted lists will be ANDed and the resulting PAS will be quite small which will in turn lead to a small communication volume. This is not valid for OR logic since when the inverted lists are ORed in an index server, the resulting PAS will be larger. However, this is not a problem in our scheme since we can limit the PAS size in OR logic without much degrading the quality of the results (Section 3.2.2, Index Server Algorithm). Thus, we can avoid large communication volumes in OR logic.

4.4. Replicated Hypergraph-partitioning-based Index Distribution

A recent study that addresses replication in hypergraphs is discussed in [20]. A tool named rpPaToH which can replicate the vertices of an undirected hypergraph to improve the cuto size of a partition significantly via utilizing a given amount of replication is developed and presented in [20].

To replicate terms (which are modeled as vertices) in the hypergraph model introduced in 4.3, we use rpPaToH. In this way, replication can directly be used to improve the objective modeled by the hypergraph partitioning. This forms the main difference of our replication approach compared to the most frequent term replication mentioned in Section 4.2. By using a hypergraph partitioning model and perform vertex replication during partitioning, we can actually address a particular objective while maintaining balance on a certain criterion (see Section 4.3 for the objective and balance constraint of the used hypergraph model in this work). Moreover, using replicated hypergraph partitioning allows us to replicate terms in a finer granularity compared to the most frequent term replication, which replicates frequent terms to all index servers. In other words, instead of replicating frequent terms to all index servers greedily, we replicate terms to the index server(s) where they are most “needed” according to the defined objective. Note that, in index servers, the replication of terms and their inverted lists has only storage overhead and does not incur any additional computational cost neither in most frequent term replication nor in rpPaToH.

Fig. 4 shows a four-way replicated partition of the hypergraph given in Fig. 3. As seen from the figure, $t_1$ is replicated in $IS_1, IS_2$, and $IS_4$, $t_4$ is replicated in
By using the flexibility replication provides, we can also of a parallel query processing system, can be reduced.

The hypergraph model tries to minimize the number of index servers to answer a query. We also propose a hybrid scheme that mediates these two extremes.

To see how crucial scheduling decisions can be in minimizing the number of index servers involved in answering a query, consider $q_2$ in Fig. 4. When the central broker is to form subqueries for this query, it can select from several options. One of them is $q_2^1 = \{t_1\}, q_2^2 = \emptyset, q_2^3 = \{t_5\}$, and $q_2^4 = \{t_7\}$, which requires three index servers to answer this query. Another approach would be to schedule $q_2^4$ as $q_2^1 = \emptyset, q_2^2 = \emptyset, q_2^3 = \emptyset$, and $q_2^4 = \{t_1, t_5, t_7\}$, which schedules all terms to $IS_4$ and thus requires only one index server for answering $q_2$.

5. Query Scheduling Heuristics

When there are replicated terms, the problem of selecting which replicas to use arises. Depending on the selection made, the scheduling and subquery forming can change dramatically. To schedule queries and form subqueries from them, we maintain a table at central broker about which terms are stored by index servers (Section 3). As mentioned in Section 4.3, the proposed hypergraph model tries to minimize the number of index servers involved in answering a query. In this way the communication volume, which is one of the most important factors that determines the overall performance of a parallel query processing system, can be reduced. By using the flexibility replication provides, we can also balance the load of the index servers. This section introduces algorithms to schedule replicated terms. The spectrum of algorithms presented here have both extremes, where in one extreme we only consider minimizing number of index servers involved in answering a query, and in the other extreme we only consider balancing the load of the index servers using dynamic information about them. We also propose a hybrid scheme that mediates these two extremes.

To see how crucial scheduling decisions can be in minimizing the number of index servers involved in answering a query, consider $q_1$ in Fig. 4. When the central broker is to form subqueries for this query, it can select from several options. One of them is $q_1^1 = \{t_1\}, q_1^2 = \emptyset, q_1^3 = \{t_5\}$, and $q_1^4 = \{t_7\}$, which requires three index servers to answer this query. Another approach would be to schedule $q_1^4$ as $q_1^1 = \emptyset, q_1^2 = \emptyset, q_1^3 = \emptyset$, and $q_1^4 = \{t_1, t_5, t_7\}$, which schedules all terms to $IS_4$ and thus requires only one index server for answering $q_2$.

5.1. Reduction to Set Cover Problem and Set-Cover-Based Scheduling

In this section, we show that minimizing number of index servers involved in answering a single query in replicated and term partitioned indexes is equivalent to the set cover problem. We follow the approach mentioned in [20] where the same problem is solved for the final cutsize computation in replicated partitioning for undirected hypergraphs. Before showing how this reduction is done, we need to eliminate the terms requested by a query which are not replicated since there is only a single instance of a non-replicated term and it is obvious which index server is going to answer this term. An immediate observation following this proposition is that, since we have to use certain index servers for non-replicated terms of a given query, we can (and should) try to select the replicas of the replicated terms occurring in this query from these index servers, if possible. In this way, we do not increase the number of index servers involved in answering the given query after scheduling non-replicated terms while choosing replicas of the replicated terms.

For example, consider $q_1 = \{t_1, t_2\}$ in Fig. 4. The $IS_1$ has to involve in answering this query since it is the only index server that stores $t_2$. However, there are three alternatives for $t_1$, which are $IS_1, IS_2$, and $IS_4$. The above-mentioned observation simply tells us to select $t_1$ from one of the index servers that are already used for answering non-replicated terms, which is $IS_1$ in this case. By doing so, we do not increase the number of index servers involved in answering this query while selecting replicas of the replicated terms of it. After elimi-
nating non-replicated terms and replicated terms whose replicas we can choose from the index servers we use for non-replicated terms, we are left only with the replicated terms which we cannot retrieve from already chosen index servers. This case can be seen for $q_2$, all of whose terms are replicated and cannot be selected from the index servers chosen for non-replicated terms.

In such cases, for the remaining replicated terms, the scheduling problem reduces to the set cover problem which is known to be NP-complete [21]. Again, consider $q_2$ in Fig. 4. If we model this query and its deselected replicated terms with the set-cover problem, the ground set for this query becomes $S = \{t_1, t_5, t_7\}$, and we want to cover this set by using minimum number of sets (which correspond to index servers) among the possible sets we can use for the selection of replicas: $S_{IS_1} = \{t_1, t_5\}, S_{IS_2} = \{t_1, t_7\}, S_{IS_3} = \{t_5\}$, and $S_{IS_4} = \{t_1, t_5, t_7\}$. Obviously, the optimum value is one, which is the selection of the set $S_{IS_4}$ (thus the index server $IS_4$) to answer $q_2$.

Based on the mentioned observation, in the set-cover-based scheduling heuristic, we first schedule the non-replicated terms of a query. Then, replicated terms of this query are checked for their replicas whether any of them has replicas in the index servers that are going to be used for the non-replicated terms. If so, these replicas are used. If there remain any replicated term(s) which could not be scheduled using the method mentioned above, the scheduling of such replicated terms can be reduced to the set cover problem. Since this problem is NP-complete, a simple heuristic [22] is adopted to obtain a schedule for the remaining replicated terms. In each iteration, this heuristic simply selects the index server that contains the largest number of uncovered replicas so far, and then removes the currently covered replicas from all index servers. This process is repeated till there remain no uncovered replicated terms. This heuristic has an approximation ratio of $\ln(n)+1$ [22], where $n$ is the total number of elements in the ground set constructed for this query.

Fig. 5 illustrates the schedule obtained using set-cover-based scheduling on the replicated partition and the queries given in Fig. 4. Note that the central broker maintains a map structure which is also shown in the figure. For $q_1 = \{t_1, t_2\}$, after selecting $IS_1$ for the non-replicated term $t_2$, since the replicated term $t_1$ can be selected from the index servers selected for the non-replicated terms, $t_1$ is scheduled to $IS_1$, and the sub-queries are formed respectively. Since $q_2 = \{t_1, t_5, t_7\}$ requires terms which are all replicated, we use the mentioned heuristic for the set cover problem. As seen in the figure, $IS_4$ contains the largest number of replicas and thus we select it. After using replicas from $IS_4$, there remain no more replicas to cover, so we are done for $q_2$.

Using the set-cover-based scheduling for the remaining two queries in a similar manner, we schedule $q_3$ to $IS_2$ and $IS_4$, and $q_4$ to $IS_3$ and $IS_4$.

5.2. Dynamic Load Balancing

In this scheme, the central broker makes the scheduling decision based on the dynamic information regarding index servers. The main idea behind dynamic load balancing is to use the flexibility while selecting the replicas of the replicated terms. Instead of making the scheduling decision to minimize the number of index servers involved in answering a query as in Section 5.1, the central broker schedules replicated query terms to the index servers with current minimum load. To do this, it is enough for the central broker to hold a simple array of size $K$, which contains information about how
many subqueries exist in the queue of each index server at a given time. Using this information, the central broker can dynamically schedule the replicas of the replicated terms of a query to minimally loaded index servers whenever possible. In this way, it may be possible to achieve a better load balance (on disk IOs, communication volume, etc.) than the set-cover-based scheduling, although it is likely that this scheme will incur a higher communication volume.

5.3. Hybrid Scheduling

Set-cover-based scheduling can suffer from load imbalance whereas dynamic load balancing can suffer from high communication volume. An idea would be to use both scheduling algorithms by assigning certain weights of importance to them. This allows us to use a mixture of these algorithms following a tradeoff between the benefits and drawbacks of them. This hybrid algorithm may have a higher communication volume and a better load balance than the set-cover-based scheduling, and a worse load balance and a lower communication volume than the dynamic load balancing, however, we expect it to have a higher throughput and lower average response time compared to both scheduling heuristics.

We assume that both the set-cover-based and dynamic scheduling heuristics provide an ordered list of index servers as output. The order of an index server in these ordered lists is called its rank. In our approach, we use the ranks of both scheduling algorithms to schedule terms of a query to index servers. For a query, let \( r_S \) and \( r_D \) be one-to-one and onto functions that return the rank of each index server for the set-cover-based and dynamic load balancing scheduling heuristics, respectively. Normally, if these functions are used standalone, both heuristics select the index server with minimum rank, update necessary data structures, and repeat this process until all terms are scheduled to index servers. To use a mixture of them, we assign a certain weight of importance to both functions to determine the rank of each index server:

\[
rank(i) = \alpha \times r_S(i) + (1 - \alpha) \times r_D(i), \quad \text{for} \ 1 \leq i \leq K,
\]

where we select the index server with the smallest rank. After selecting an index server, necessary data structures are updated and this process is repeated till all terms of the given query are scheduled. If \( \alpha = 1 \), hybrid scheduling is equivalent to the set-cover-based scheduling, whereas if \( \alpha = 0 \), it is equivalent to dynamic load balancing. In our experiments, we identify good \( \alpha \) values for both AND and OR logic empirically.

6. Experimental Results
References


Table 2: Average Response Time (ART), Throughput (THR), and Average Number of Processors (ANP) statistics for Round-Robin (RR), Robin-Robin+MF (RR+MF) (10\%), Robin-Robin+MF (25\%), PaToH, rpPaToH (10\%), and rpPaToH (25\%) with varying $\alpha$ values for the AND-based query retrieval for $K = 8$.

<table>
<thead>
<tr>
<th>Partitioning/Replication Scheme</th>
<th>$\alpha$</th>
<th>RR</th>
<th>RR+MF (10%)</th>
<th>RR+MF (25%)</th>
<th>PaToH</th>
<th>rpPaToH (10%)</th>
<th>rpPaToH (25%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ART</td>
<td>1.0</td>
<td>2.678</td>
<td>0.897</td>
<td>0.616</td>
<td>2.276</td>
<td>0.619</td>
<td>0.409</td>
</tr>
<tr>
<td>THR</td>
<td>18.6</td>
<td>55.6</td>
<td>80.6</td>
<td>21.9</td>
<td>80.6</td>
<td>120.5</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>1.98</td>
<td>1.69</td>
<td>1.89</td>
<td>1.37</td>
<td>1.15</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.9</td>
<td>2.681</td>
<td>0.891</td>
<td>0.615</td>
<td>2.261</td>
<td>0.616</td>
<td>0.410</td>
</tr>
<tr>
<td>THR</td>
<td>18.6</td>
<td>55.9</td>
<td>80.8</td>
<td>22.1</td>
<td>80.2</td>
<td>121.2</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>1.98</td>
<td>1.69</td>
<td>1.89</td>
<td>1.37</td>
<td>1.14</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.8</td>
<td>2.696</td>
<td>0.913</td>
<td>0.613</td>
<td>2.253</td>
<td>0.654</td>
<td>0.425</td>
</tr>
<tr>
<td>THR</td>
<td>18.4</td>
<td>54.4</td>
<td>81.0</td>
<td>22.1</td>
<td>76.0</td>
<td>117.0</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.00</td>
<td>1.71</td>
<td>1.89</td>
<td>1.39</td>
<td>1.18</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.7</td>
<td>2.664</td>
<td>0.989</td>
<td>0.656</td>
<td>2.268</td>
<td>0.716</td>
<td>0.465</td>
</tr>
<tr>
<td>THR</td>
<td>18.7</td>
<td>50.4</td>
<td>75.9</td>
<td>21.9</td>
<td>68.9</td>
<td>107.2</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.04</td>
<td>1.75</td>
<td>1.89</td>
<td>1.44</td>
<td>1.24</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.6</td>
<td>2.689</td>
<td>1.069</td>
<td>0.725</td>
<td>2.253</td>
<td>0.798</td>
<td>0.541</td>
</tr>
<tr>
<td>THR</td>
<td>18.5</td>
<td>46.5</td>
<td>68.6</td>
<td>22.1</td>
<td>62.1</td>
<td>91.9</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.08</td>
<td>1.80</td>
<td>1.89</td>
<td>1.50</td>
<td>1.32</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.5</td>
<td>2.680</td>
<td>1.174</td>
<td>0.798</td>
<td>2.250</td>
<td>0.932</td>
<td>0.664</td>
</tr>
<tr>
<td>THR</td>
<td>18.6</td>
<td>42.5</td>
<td>62.3</td>
<td>22.2</td>
<td>53.2</td>
<td>74.7</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.13</td>
<td>1.86</td>
<td>1.89</td>
<td>1.59</td>
<td>1.44</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.4</td>
<td>2.679</td>
<td>1.264</td>
<td>0.887</td>
<td>2.267</td>
<td>1.089</td>
<td>0.833</td>
</tr>
<tr>
<td>THR</td>
<td>18.5</td>
<td>39.3</td>
<td>55.8</td>
<td>21.9</td>
<td>45.5</td>
<td>59.3</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.18</td>
<td>1.94</td>
<td>1.89</td>
<td>1.70</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.3</td>
<td>2.671</td>
<td>1.396</td>
<td>1.011</td>
<td>2.247</td>
<td>1.327</td>
<td>1.055</td>
</tr>
<tr>
<td>THR</td>
<td>18.6</td>
<td>35.7</td>
<td>49.1</td>
<td>22.2</td>
<td>37.5</td>
<td>47.1</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.61</td>
<td>2.26</td>
<td>2.03</td>
<td>1.89</td>
<td>1.81</td>
<td>1.72</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.2</td>
<td>2.708</td>
<td>1.596</td>
<td>1.172</td>
<td>2.258</td>
<td>1.534</td>
<td>1.275</td>
</tr>
<tr>
<td>THR</td>
<td>18.4</td>
<td>31.2</td>
<td>42.4</td>
<td>22.1</td>
<td>32.4</td>
<td>38.9</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.61</td>
<td>2.35</td>
<td>2.13</td>
<td>1.89</td>
<td>1.90</td>
<td>1.84</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.1</td>
<td>2.706</td>
<td>1.727</td>
<td>1.271</td>
<td>2.230</td>
<td>1.686</td>
<td>1.454</td>
</tr>
<tr>
<td>THR</td>
<td>18.4</td>
<td>28.9</td>
<td>39.2</td>
<td>22.4</td>
<td>29.6</td>
<td>34.2</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.42</td>
<td>2.21</td>
<td>1.89</td>
<td>1.95</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>ART</td>
<td>0.0</td>
<td>2.677</td>
<td>1.740</td>
<td>1.277</td>
<td>2.249</td>
<td>1.674</td>
<td>1.437</td>
</tr>
<tr>
<td>THR</td>
<td>18.6</td>
<td>28.6</td>
<td>39.0</td>
<td>22.2</td>
<td>29.7</td>
<td>34.6</td>
<td></td>
</tr>
<tr>
<td>ANP</td>
<td>2.60</td>
<td>2.42</td>
<td>2.21</td>
<td>1.89</td>
<td>1.94</td>
<td>1.91</td>
<td></td>
</tr>
</tbody>
</table>
Table 3: Average Response Time (ART), Throughput (THR), and Average Number of Processors (ANP) statistics for Round-Robin (RR), Robin-Robin+MF (RR+MF) (10%), Robin-Robin+MF (25%), PaToH, rpPaToH (10%), and rpPaToH (25%) with varying $\alpha$ values for the AND-based query retrieval for $K = 16$.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>ART</th>
<th>RR</th>
<th>RR+MF (10%)</th>
<th>RR+MF (25%)</th>
<th>PaToH</th>
<th>rpPaToH (10%)</th>
<th>rpPaToH (25%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.177</td>
<td>0.857</td>
<td>0.565</td>
<td>1.682</td>
<td>0.539</td>
<td>0.353</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.9</td>
<td>58.1</td>
<td>88.0</td>
<td>29.6</td>
<td>91.9</td>
<td>139.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.81</td>
<td>2.03</td>
<td>2.01</td>
<td>1.56</td>
<td>1.34</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>2.199</td>
<td>0.876</td>
<td>0.572</td>
<td>1.695</td>
<td>0.552</td>
<td>0.353</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.7</td>
<td>56.8</td>
<td>86.6</td>
<td>29.2</td>
<td>89.5</td>
<td>138.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.82</td>
<td>2.04</td>
<td>2.01</td>
<td>1.58</td>
<td>1.34</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>2.194</td>
<td>0.930</td>
<td>0.616</td>
<td>1.692</td>
<td>0.572</td>
<td>0.389</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.6</td>
<td>53.5</td>
<td>80.5</td>
<td>29.3</td>
<td>86.2</td>
<td>126.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.84</td>
<td>2.07</td>
<td>2.01</td>
<td>1.60</td>
<td>1.39</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>2.163</td>
<td>0.993</td>
<td>0.683</td>
<td>1.680</td>
<td>0.644</td>
<td>0.442</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23.1</td>
<td>49.8</td>
<td>72.8</td>
<td>29.6</td>
<td>76.6</td>
<td>110.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.86</td>
<td>2.12</td>
<td>2.01</td>
<td>1.65</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>2.184</td>
<td>1.038</td>
<td>0.729</td>
<td>1.678</td>
<td>0.714</td>
<td>0.487</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.8</td>
<td>47.9</td>
<td>68.0</td>
<td>29.6</td>
<td>69.3</td>
<td>101.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.88</td>
<td>2.16</td>
<td>2.01</td>
<td>1.71</td>
<td>1.53</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>2.199</td>
<td>1.086</td>
<td>0.787</td>
<td>1.692</td>
<td>0.781</td>
<td>0.559</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.7</td>
<td>45.7</td>
<td>63.1</td>
<td>29.4</td>
<td>63.3</td>
<td>88.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.90</td>
<td>2.22</td>
<td>2.01</td>
<td>1.78</td>
<td>1.63</td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td>2.192</td>
<td>1.170</td>
<td>0.849</td>
<td>1.700</td>
<td>0.896</td>
<td>0.678</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.7</td>
<td>42.5</td>
<td>58.6</td>
<td>29.1</td>
<td>55.3</td>
<td>73.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.92</td>
<td>2.27</td>
<td>2.01</td>
<td>1.87</td>
<td>1.75</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>2.180</td>
<td>1.224</td>
<td>0.930</td>
<td>1.681</td>
<td>1.024</td>
<td>0.814</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.7</td>
<td>40.5</td>
<td>53.5</td>
<td>29.6</td>
<td>48.5</td>
<td>60.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.94</td>
<td>2.34</td>
<td>2.01</td>
<td>1.96</td>
<td>1.87</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>2.170</td>
<td>1.323</td>
<td>1.027</td>
<td>1.668</td>
<td>1.161</td>
<td>0.987</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.9</td>
<td>37.6</td>
<td>48.5</td>
<td>29.8</td>
<td>42.7</td>
<td>50.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.96</td>
<td>2.42</td>
<td>2.01</td>
<td>2.04</td>
<td>1.99</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>2.145</td>
<td>1.429</td>
<td>1.132</td>
<td>1.693</td>
<td>1.328</td>
<td>1.198</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23.2</td>
<td>34.8</td>
<td>43.8</td>
<td>29.5</td>
<td>37.5</td>
<td>41.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.98</td>
<td>2.50</td>
<td>2.01</td>
<td>2.11</td>
<td>2.11</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>2.192</td>
<td>1.477</td>
<td>1.176</td>
<td>1.681</td>
<td>1.388</td>
<td>1.263</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.7</td>
<td>33.6</td>
<td>42.3</td>
<td>29.6</td>
<td>35.8</td>
<td>39.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.80</td>
<td>3.99</td>
<td>2.54</td>
<td>2.01</td>
<td>2.13</td>
<td>2.13</td>
<td></td>
</tr>
</tbody>
</table>
Table 4: Average Response Time (ART), Throughput (THR), and Average Number of Processors (ANP) statistics for Round-Robin (RR), Robin-Robin+MF (RR+MF) (10%), Robin-Robin+MF (RR+MF) (25%), PaToH, rpPaToH (10%), and rpPaToH (25%) with varying α values for the OR-based query retrieval for $K = 8$.

<table>
<thead>
<tr>
<th>α</th>
<th>ART (RR)</th>
<th>ART (RR+MF 10%)</th>
<th>ART (RR+MF 25%)</th>
<th>ART (PaToH)</th>
<th>ART (rpPaToH 10%)</th>
<th>ART (rpPaToH 25%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.768</td>
<td>0.665</td>
<td>0.620</td>
<td>0.772</td>
<td>0.572</td>
<td>0.482</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64.7</td>
<td>74.7</td>
<td>80.3</td>
<td>64.2</td>
<td>86.4</td>
<td>103.4</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.60</td>
<td>1.98</td>
<td>1.69</td>
<td>1.89</td>
<td>1.37</td>
<td>1.15</td>
</tr>
<tr>
<td>0.9</td>
<td>ART</td>
<td>0.766</td>
<td>0.654</td>
<td>0.627</td>
<td>0.779</td>
<td>0.577</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.9</td>
<td>75.7</td>
<td>79.3</td>
<td>64.0</td>
<td>86.3</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>1.98</td>
<td>1.69</td>
<td>1.89</td>
<td>1.37</td>
</tr>
<tr>
<td>0.8</td>
<td>ART</td>
<td>0.772</td>
<td>0.624</td>
<td>0.586</td>
<td>0.784</td>
<td>0.537</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.3</td>
<td>79.8</td>
<td>84.8</td>
<td>63.2</td>
<td>92.6</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>1.99</td>
<td>1.70</td>
<td>1.89</td>
<td>1.40</td>
</tr>
<tr>
<td>0.7</td>
<td>ART</td>
<td>0.777</td>
<td>0.582</td>
<td>0.545</td>
<td>0.775</td>
<td>0.513</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.0</td>
<td>85.3</td>
<td>91.1</td>
<td>64.3</td>
<td>96.9</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.01</td>
<td>1.72</td>
<td>1.89</td>
<td>1.43</td>
</tr>
<tr>
<td>0.6</td>
<td>ART</td>
<td>0.771</td>
<td>0.583</td>
<td>0.526</td>
<td>0.774</td>
<td>0.505</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.5</td>
<td>85.2</td>
<td>94.5</td>
<td>64.0</td>
<td>98.4</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.04</td>
<td>1.75</td>
<td>1.89</td>
<td>1.49</td>
</tr>
<tr>
<td>0.5</td>
<td>ART</td>
<td>0.767</td>
<td>0.586</td>
<td>0.523</td>
<td>0.779</td>
<td>0.509</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.9</td>
<td>84.8</td>
<td>94.9</td>
<td>64.0</td>
<td>97.6</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.10</td>
<td>1.83</td>
<td>1.88</td>
<td>1.60</td>
</tr>
<tr>
<td>0.4</td>
<td>ART</td>
<td>0.777</td>
<td>0.597</td>
<td>0.541</td>
<td>0.769</td>
<td>0.528</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>63.9</td>
<td>83.2</td>
<td>91.8</td>
<td>64.5</td>
<td>94.0</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.17</td>
<td>1.92</td>
<td>1.89</td>
<td>1.71</td>
</tr>
<tr>
<td>0.3</td>
<td>ART</td>
<td>0.776</td>
<td>0.605</td>
<td>0.553</td>
<td>0.775</td>
<td>0.553</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.0</td>
<td>82.1</td>
<td>89.8</td>
<td>64.2</td>
<td>89.8</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.25</td>
<td>2.03</td>
<td>1.89</td>
<td>1.82</td>
</tr>
<tr>
<td>0.2</td>
<td>ART</td>
<td>0.778</td>
<td>0.630</td>
<td>0.577</td>
<td>0.780</td>
<td>0.568</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>63.9</td>
<td>78.8</td>
<td>86.1</td>
<td>63.5</td>
<td>87.3</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.35</td>
<td>2.13</td>
<td>1.89</td>
<td>1.89</td>
</tr>
<tr>
<td>0.1</td>
<td>ART</td>
<td>0.770</td>
<td>0.644</td>
<td>0.596</td>
<td>0.776</td>
<td>0.581</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>64.6</td>
<td>77.1</td>
<td>83.3</td>
<td>64.3</td>
<td>85.6</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.60</td>
<td>2.41</td>
<td>2.21</td>
<td>1.89</td>
<td>1.95</td>
</tr>
<tr>
<td>0.0</td>
<td>ART</td>
<td>0.761</td>
<td>0.639</td>
<td>0.593</td>
<td>0.773</td>
<td>0.583</td>
</tr>
<tr>
<td></td>
<td>THR</td>
<td>65.3</td>
<td>77.8</td>
<td>83.6</td>
<td>64.0</td>
<td>85.2</td>
</tr>
<tr>
<td></td>
<td>ANP</td>
<td>2.61</td>
<td>2.41</td>
<td>2.21</td>
<td>1.89</td>
<td>1.95</td>
</tr>
</tbody>
</table>
Table 5: Average Response Time (ART), Throughput (THR), and Average Number of Processors (ANP) statistics for Round-Robin (RR), Robin-Robin+MF (RR+MF (10%)), Robin-Robin+MF (25%), PaToH, rpPaToH (10%), and rpPaToH (25%) with varying $\alpha$ values for the OR-based query retrieval for $K = 16$.

| $\alpha$ | Partitioning/Replication Scheme | ART | THR | ANP | ART | THR | ANP | ART | THR | ANP | ART | THR | ANP | ART | THR | ANP | ART | THR | ANP | ART | THR | ANP | ART | THR | ANP |
|---------|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1.0     | RR                              | 0.471 | 0.396 | 0.386 | 0.422 | 0.328 | 0.302 | RR+MF (10%) | 0.468 | 0.380 | 0.372 | 0.424 | 0.321 | 0.293 | RR+MF (25%) | 0.469 | 0.373 | 0.342 | 0.426 | 0.306 | 0.278 | PaToH | 0.470 | 0.371 | 0.339 | 0.423 | 0.302 | 0.269 |
| 0.9     | RR                              | 0.105.2 | 124.9 | 127.5 | 117.2 | 150.6 | 163.4 | RR+MF (10%) | 0.106.2 | 130.0 | 132.8 | 116.6 | 153.4 | 168.8 | RR+MF (25%) | 0.105.9 | 132.3 | 144.3 | 115.4 | 161.4 | 177.2 | PaToH | 0.105.4 | 132.9 | 145.0 | 116.6 | 162.6 | 183.0 |
| 0.8     | RR                              | 0.105.9 | 132.6 | 146.2 | 116.3 | 162.1 | 183.3 | RR+MF (10%) | 0.104.9 | 132.6 | 146.2 | 116.3 | 162.1 | 183.3 | RR+MF (25%) | 0.104.9 | 132.6 | 146.2 | 116.3 | 162.1 | 183.3 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
|        | RR                              | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (10%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | RR+MF (25%) | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 | PaToH | 0.105.8 | 132.9 | 146.0 | 114.6 | 160.5 | 175.8 |
REPLICATED HYPERGRAPH PARTITIONING

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Reha Oğuz Selvitopi
September, 2010
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

__________________________
Prof. Dr. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

__________________________
Asst. Prof. Dr. Özcan Öztürk

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

__________________________
Assoc. Prof. Dr. Oya Ekin Karaşan

Approved for the Institute of Engineering and Science:

__________________________
Prof. Dr. Levent Onural
Director of the Institute
ABSTRACT

REPLICATED HYPERGRAPH PARTITIONING

Reha Oğuz Selvitopi
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2010

Hypergraph partitioning is recently used in distributed information retrieval (IR) and spatial databases to correctly capture the communication and disk access costs. In the hypergraph models for these areas, the quality of the partitions obtained using hypergraph partitioning can be crucial for the objective of the targeted problem. Replication is a widely used terminology to address different performance issues in distributed IR and database systems. The main motivation behind replication is to improve the performance of the targeted issue at the cost of using more space.

In this work, we focus on replicated hypergraph partitioning schemes that improve the quality of hypergraph partitioning by vertex replication. To this end, we propose a replicated partitioning scheme where replication and partitioning are performed in conjunction. Our approach utilizes successful multilevel and recursive bipartitioning methodologies for hypergraph partitioning. The replication is achieved in the uncoarsening phase of the multilevel methodology by extending the efficient Fiduccia-Mattheyses (FM) iterative improvement heuristic. We call this extended heuristic replicated FM (rFM). The proposed rFM heuristic supports move, replication and unreplication operations on the vertices by introducing new algorithms and vertex states. We show rFM has the same complexity as FM and integrate the proposed replication scheme into the multilevel hypergraph partitioning tool PaToH. We test the proposed replication scheme on realistic datasets and obtain promising results.

Keywords: Hypergraph partitioning, data replication, iterative improvement heuristics.
OZET
ÇOKLAMALI HİPERÇİZGE BÖLÜMLEME

Reha Oğuz Selvitopi
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2010

Hiperçizge bölütleme son zamanlarda dağıtık veri erişimi ve uzamsal veri tabanlarında iletişim ve disk erişim maliyetlerini doğru bir şekilde yakalamak için kullanılmıştır. Bu alanlardaki hiperçizge modellerinde, hiperçizge bölütleme kullanılarak elde edilen bölümlerin kalitesi hedeflenen problemin objektifi için çok önemli olabilir. Çoklama, dağıtık veri erişimi ve veri taban sistemlerinde çeşitli performans meselerini ele almak için yaygın olarak kullanılan bir terminolojidir. Çoklamının arkasındaki ana motivasyon, hedeflenen konunun performansını daha fazla alan kullanma pahasına geliştirmektir.

Bu çalışmada, hiperçizge bölütlemenin kalitesini düğüm çoklamasıyla geliştirilen hiperçizge bölütleme şemalarının üstüne odaklanıyoruz. Bu aşamada, çoklama ve bölütlemenin bir arada yapıldığı bir çoklama hiperçizge bölütleme şemasi öneriyoruz. Yaklaşımımız, hiperçizge bölütlemesi için başarılı çok seviyeli ve özünelemeli ikiye bölütleme yöntemlerini kullanmaktadır. Çoklama, çok seviyeli yöntemin açılma safhasında verimli Fiduccia-Mattheyses (FM) yinelemeli geliştirme sezgiselini genişletecek şekilde edilmektedir. Bu genişletilmiş versiyona çoklama FM (rFM) diyoruz. Önerilen rFM sezgisi yeni algoritmalar ve köşe durumları öne sürekten taşımak, çoklama ve azlama işlemlerini desteklemektedir. Önerilen çoklama şemasını çok seviyeli hiperçizge bölütleme aracı PaToH'a entegre edip çeşitli gerçekçi veri takımları üstünde test ediyoruz.

Anahtar sözcükler: Hiperçizge bölütleme, veri çoklama, yinelemeli geliştirme sezgiselleri.
Acknowledgement

I would like to thank to my thesis supervisor Prof. Dr. Cevdet Aykanat for his valuable suggestions, support and guidance throughout the development of this thesis.

I am grateful to Asst. Prof. Dr. Özcan Öztürk and Assoc. Prof. Dr. Oya Ekin Karaşan for reading, commenting and sharing their ideas on the thesis.

I am specially thankful to Ata Türk for sharing his ideas, suggestions and valuable experiences with me throughout the year. I thank Volkan Yazıcı for his comments and support whose endless desire to learn has always inspired me.

I am grateful to my family and their infinite understanding and support throughout my life.
Contents

1 Introduction 1

2 Background and Problem Definition 4
   2.1 Definitions and Hypergraph Partitioning Problem . . . . . . . . 4
   2.2 Iterative Improvement Heuristics for HP . . . . . . . . . . . . . 5
   2.3 Multilevel and Recursive Bipartitioning Frameworks . . . . . . 6
   2.4 Motivation and Replicated HP Problem . . . . . . . . . . . . . . 8

3 Related Work 9

4 Replicated Hypergraph Partitioning 13
   4.1 Replicated FM (rFM) . . . . . . . . . . . . . . . . . . . . . . . . . 14
      4.1.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 14
      4.1.2 Overall rFM Algorithm . . . . . . . . . . . . . . . . . . . . 17
      4.1.3 Initial Gain Computation and Gain Update Algorithms . . 20
   4.2 rFM and Multilevel Framework . . . . . . . . . . . . . . . . . . . 37

vi
CONTENTS

4.3 Recursive Bipartitioning and Replica Selection ............... 39
   4.3.1 Cut-net Splitting .................................................. 39
   4.3.2 Replica Selection .................................................. 41
   4.3.3 Replication Amount Distribution .............................. 43

5 Experimental Results ................................................. 44
   5.1 Experimental Setup .................................................. 44
   5.2 Datasets .............................................................. 46
   5.3 Results ............................................................... 47

6 Conclusion and Future Work ........................................ 59
List of Figures

4.1 Move and replication of a vertex. ........................................ 15
4.2 Unreplication of a replicated vertex. ................................. 16
4.3 A net with no non-replicated vertices can be internal to any part. 22
4.4 Various unreplication operations on the replicas of a net that has
no non-replicated vertices. ............................................. 23
4.5 Initial gains and pin distributions. .................................... 25
4.6 Gains and pin distributions after moving $v_4$. ..................... 28
4.7 Gains and pin distributions after replicating $v_6$. .................. 32
4.8 Gains and pin distributions after unreplicating $v_1$ from $V_B$ .... 35
4.9 Cut-net splitting, no pins of net $n_j$ are discarded. ............. 40
4.10 Cut-net splitting, the pins of net $n_j$ in part $A$ are discarded. . . 40
4.11 Three replica selection alternatives for $n_j$. ....................... 42
4.12 Replica selection for $n_j$ ($\lambda_j = 3$) and $n_k$ ($\lambda_k = 2$). .... 43
5.1 Improvement at $K = 32$. .................................................. 52
5.2 Improvement at $K = 64$. .................................................. 53
5.3 Improvement at $K = 128$. . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.4 Improvement at $K = 256$. . . . . . . . . . . . . . . . . . . . . . . . . . . 54
List of Tables

5.1 The properties of the data sets used in our experiments. . . . . . . 46
5.2 The cut and imbalance values for $K = 32$. . . . . . . . . . . . . . 48
5.3 The cut and imbalance values for $K = 64$. . . . . . . . . . . . . . 49
5.4 The cut and imbalance values for $K = 128$. . . . . . . . . . . . . . 50
5.5 The cut and imbalance values for $K = 256$. . . . . . . . . . . . . . 51
5.6 The cut values for rFM and gradient rFM for $K = 32$. . . . . . . . 55
5.7 The cut values for rFM and gradient rFM for $K = 64$. . . . . . . . 56
5.8 The cut values for rFM and gradient rFM for $K = 128$. . . . . . . . 57
5.9 The cut values for rFM and gradient rFM for $K = 256$. . . . . . . 58
Chapter 1

Introduction

There are various models that uses hypergraph partitioning for different objectives in different fields such as parallel scientific computing [4, 14, 24, 60], VLSI circuit design [2, 43], distributed IR [13] and database systems [21, 22, 44, 39]. In the areas where hypergraph partitioning (HP) is used, the hypergraph models can broadly be classified into two categories as directional and undirectional hypergraph models. Generally, hypergraph models in parallel scientific computing and VLSI circuit design fall into directional models, whereas hypergraph models in distributed IR and database systems fall into undirectional models.

Recently, undirectional hypergraph models are successfully used to address the issues in distributed information retrieval (IR) [13] and spatial databases [21, 22]. In distributed IR, hypergraph models are used to reduce the communication volume and improve the load balance. In spatial databases, the disk access costs can be reduced for aggregate queries by using hypergraph models. In the hypergraph models for both areas, improving the quality of the partitions obtained using HP by reducing the cutsize is crucial for the problem.

Replication is a widely used terminology in various computer science fields such as distributed IR [5, 11, 47, 48, 50, 57] and database systems [6, 7, 29, 61]. The basic purpose of replication differs from field to field. Generally, replication is used to improve the performance of the target system by reducing the costs
of different objectives at the expense of using more space. Replication is a valuable tool in distributed IR systems to improve the query throughput and fault tolerance. In database systems, the records in the database are replicated across multiple sites to improve the performance of the read operations.

In this work, we propose a replication scheme for hypergraph partitioning that aims to improve the quality of the partitions by replicating vertices that uses undirectional hypergraph models. To our knowledge, this problem has not been addressed for undirectional hypergraph models. In the context of directional hypergraph models, especially in VLSI literature, this is a well-studied problem [40, 33, 34, 45, 63, 26]. In VLSI circuit partitioning, the replication of a vertex in a partitioned circuit may bring internal nets to the cut which are connected to that vertex where input pins of the source vertex need to be replicated along with the replicated vertex since the proposed hypergraph models are directional. In our replication scheme, the replication of a vertex cannot bring any internal net to the cut, i.e., replication cannot increase the cutsize of a bipartition. This forms the basic difference between the replication schemes for directional and undirectional hypergraph models.

Our approach is a single-phase methodology that performs replication along with the partitioning. It uses multilevel and recursive bipartitioning frameworks for HP. We achieve replication in the uncoarsening phase of the multilevel methodology by using a refinement heuristic as a replication tool. We extend the Fiduccia-Mattheyses (FM) heuristic [28] to be capable of replication and unreplication of vertices in addition to standard move operation. We call this extended heuristic replicated FM (rFM). The proposed heuristic operates on a given bipartition and introduces new vertex states and gain update algorithms in order to support replication and unreplication. To obtain multi-way partitions, we adopt recursive bipartitioning methodology. The proposed replication scheme is implemented and integrated into the state–of–the–art HP tool PaToH [15]. In a concurrent work, a two-phase approach [64] is investigated where replication is performed after obtaining a $K$-way partitioning.

This thesis is organized as follows. Chapter 2 gives the necessary background
for this work and describes replicated HP problem. The previous works regarding replication in various areas are investigated in Chapter 3. Chapter 4 describes our methodology for solving replicated HP problem. We give the experimental results in Chapter 5 and conclude our work in Chapter 6.
Chapter 2

Background and Problem

Definition

2.1 Definitions and Hypergraph Partitioning Problem

A hypergraph $H = (V, N)$ is defined as a set of vertices $V$ and a set of nets $N$. Each net $n_j \in N$ connects a subset of vertices. The vertices connected by net $n_j$ are called its pins, and denoted as $Pins(n_j)$. The connection between a net $n_j$ and vertex $v_i$ is referred to as a pin $(n_j, v_i)$ of this net. The degree of a net $n_j$ is equal to the number of its pins, $|Pins(n_j)|$. The set of nets that connect vertex $v_i$ is denoted as $Nets(v_i)$ and the size of this set is equal to $|Nets(v_i)|$. Each vertex has a weight associated with it, $w(v_i)$, and each net has a cost value, $c(n_j)$. The cost function for a net is easily extended for a subset of nets $M \subseteq N$ where $c(M) = \sum_{n_j \in M} c(n_j)$.

$\Pi = \{V_1, \ldots, V_K\}$ is a $K$-way partition of $H = (V, N)$ if $V_k \neq \emptyset$ for $1 \leq k \leq K$, and $V_k \cap V_l = \emptyset$ for $1 \leq k < l \leq K$, and $\bigcup_{k=1}^{K} V_k = V$. A net is said to connect a part if it connects at least one pin in that part. Connectivity set $\Lambda_i$ of a net is defined as the set of parts connected by that net. The number of parts in the
connectivity set of \( n_j \) is denoted by \( \lambda_j = |\Lambda_j| \). A net is said to be cut if it connects more than one part \( (\lambda_j > 1) \), and uncut if it connects only one part \( (\lambda_j = 1) \). The weight \( W(V_k) \) of a part \( V_k \) is simply the sum of the weights of the vertices in that part.

The cutsize of the partition is given by

\[
\text{cutsize}(\Pi) = \sum_{n_j \in \mathcal{N}} (\lambda_j - 1)c(n_j).
\] (2.1)

This is also known as connectivity cutsize metric widely used in VLSI [43, 18] and scientific applications [14, 59, 4].

**PROBLEM 1. Hypergraph Partitioning.** Given a hypergraph \( \mathcal{H} = (\mathcal{V}, \mathcal{N}) \) and an imbalance value \( \epsilon \), find a \( K \)-way partition \( \Pi = \{V_1, \ldots, V_K\} \) that minimizes the cutsize in Equation 2.1 such that \((1+\epsilon)W_{avg} \leq W_{max}\) where, \( W_{avg} = W(\mathcal{V})/K \) and \( W_{max} = \max\{W(V_k)\} \) for \( k = 1, \ldots, K \).

This problem is known to be NP-hard [43].

## 2.2 Iterative Improvement Heuristics for HP

There are a number of algorithms based on iterative improvement heuristics for solving the HP problem. These heuristics are generally applied to iteratively improve the quality of a random initial partition. An excellent detailed discussion of these techniques can be found in [2]. Most of these algorithms are based on Fiduccia-Mattheyses (FM) [28] and Kernighan-Lin (KL) [38] heuristics, which are designed to improve the cutsize of a bipartition. KL-based heuristics achieve this by swapping vertices from the two parts of the bipartition, while FM-based heuristics use vertex moves from one part to the other. Even though KL-based heuristics perform slightly better in reducing the cutsize, FM-based heuristics are widely used due to their better running-time performance.

In FM-based heuristics, the gain of a vertex is defined as the change in the cutsize of the partition if that vertex were to be moved to its complementary
part in a bipartition. FM heuristics perform multiple passes over all vertices, where each pass comprises of a number of iterations. At the beginning of a pass, all vertices are unlocked. At each iteration of a pass, the vertex with the highest gain value is moved, locked and gain values of its unlocked neighbors are updated. At the end of each iteration, the improvement in the cutsize is stored. A pass terminates when all vertices become locked or there is no feasible move according to balance constraint. At the end of a pass, the bipartition that resulted with the minimum cutsize is restored. Multiple passes can be performed until the improvement in the cutsize drops below a certain threshold. Usually, buckets or heaps are used to store gain values.

The quality of the bipartitions produced by FM heuristic can further be improved at the expense of higher running time. The look-ahead feature and gain vectors are introduced [41] if there are more than one vertex with the highest gain which means a tie-break will occur to select the vertex to move. Different tie-breaking strategies and data structures for gains are investigated [31] and it is shown how the choices can affect the quality of the partitions found by the algorithm. There are also other ways such as using a probabilistic gain computation [25] or adding compaction to FM [55]. On the other hand, FM can be made to run faster [1] with a couple of simple techniques: (i) stop if it is unlikely to make further improvement in a pass; (ii) initialize gains only in the first pass, by rolling back the changes at the end of each pass; and (iii) use only boundary vertices to move, thus reducing the number of vertices to operate on greatly.

2.3 Multilevel and Recursive Bipartitioning Frameworks

KL and FM-based heuristics perform poorly on hypergraphs with high net degrees and they are sensitive to the quality of the initial partition. To alleviate these problems, in 1990s, multilevel algorithms are proposed [10, 32] and successfully applied to HP problem. The basic idea behind the multilevel framework is to perform a sequence of coarsening operations on the original hypergraph to obtain
a coarser hypergraph with small net degrees over which FM heuristics perform particularly better.

Multilevel methodology consists of 3 phases: coarsening, initial partitioning, and uncoarsening. In the coarsening phase, $\mathcal{H}_i = (\mathcal{V}_i, \mathcal{N}_i)$ is coarsened into $\mathcal{H}_{i+1} = (\mathcal{V}_{i+1}, \mathcal{N}_{i+1})$ for $i = 0, \ldots, m$. This is achieved by clustering vertices in $\mathcal{H}_i$ where each of these clusters becomes a vertex in $\mathcal{H}_{i+1}$. Starting from the initial hypergraph $\mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$, this coarsening operation is iteratively applied to obtain $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_m$ where $|\mathcal{V}_0| > |\mathcal{V}_1| > \ldots > |\mathcal{V}_m|$. Coarsening phase ends when the number of vertices in the coarsest hypergraph drops below a predetermined value. In the initial partitioning phase, a bipartition $\Pi_m$ of the coarsest hypergraph $\mathcal{H}_m$ is obtained. Since the coarsest hypergraph is small, the initial bipartitioning algorithm can be run a couple of times and the best bipartition can be selected. The uncoarsening phase contains exactly same number of levels as the coarsening phase. At each level of the uncoarsening, the bipartition $\Pi_i$ on $\mathcal{H}_i$ is projected back to $\Pi_{i-1}$ on $\mathcal{H}_{i-1}$. Each vertex in $\mathcal{H}_i$ is decomposed into its forming vertices in $\mathcal{H}_{i-1}$ in the projection. A finer hypergraph will have more degrees of freedom with respect to its coarser counterpart, meaning that the partition can further be improved. This is achieved by the refinement heuristics discussed in Section 2.2.

Recursive Bipartitioning (RB) is the most commonly used method for obtaining $K$-way partitions of hypergraphs although there are other methods that are based on direct $K$-way partitioning as in [36, 3]. In the RB paradigm, the initial hypergraph is bipartitioned into two new hypergraphs and then, these two new hypergraphs are further bipartitioned in a recursive manner. This procedure continues until reaching the desired number of parts in $\lg_2 K$ steps. Cut-net splitting scheme [14] is used in order to capture the connectivity cutsize metric in Equation 2.1.
2.4 Motivation and Replicated HP Problem

In this study, we focus on the HP problem with vertex replication. We refer to this problem as replicated hypergraph partitioning problem. Even though a variant of this problem arises in VLSI literature as will be explained in Related Work chapter, our focus is replication in distributed information retrieval and spatial databases. The hypergraph models are used in distributed IR [13] for improving load balance and reducing communication cost. In spatial databases [21, 22], the hypergraph models are used for reducing total disk access cost. In this work, we investigate the effects of replication in the hypergraph models for the mentioned areas where the vertices are replicated. In these areas, replication can help in improving the quality of the partitions by reducing the cutsize at the cost of using more physical space. Although we address the problems in distributed IR and spatial databases, our replication scheme can be used for any area where HP is used as a methodology and the used hypergraph model is undirectional. In our replication scheme, the replication of a vertex should not bring any internal net to the cut as opposed to the replication schemes in VLSI literature. The problem is formulated as follows:

PROBLEM 2. Replicated Hypergraph Partitioning. Given an $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, an imbalance value $\epsilon$, and a replication ratio $\rho$, find $K$ covering subsets of $\mathcal{V}$, $\Pi^R = \{V_1, \ldots, V_K\}$ such that $\bigcup_{k=1}^{K} V_k = \mathcal{V}$, $\sum_{k=1}^{K} W(V_k) \leq (1 + \rho) W(\mathcal{V})$ and $(1 + \epsilon) W_{avg} \leq W_{max}$ where, $W_{avg} = (1 + \rho) W(\mathcal{V}) / K$ and $W_{max} = \max \{W_k\}$ for $k = 1, \ldots, K$. 
Chapter 3

Related Work

Replication is a widely used term in various disciplines of computer science. Specifically, we investigate the replication schemes in VLSI literature, distributed database systems, distributed information retrieval (IR) and spatial databases.

First replication schemes in VLSI circuit design and partitioning arise in the form of gate replication to reduce pin counts and improve the cutsize of the partitioned circuits. In this respect, there are two main approaches in VLSI literature: iterative improvement based replication heuristics that generally use an extended version of the FM heuristic, and graph theoretical approaches that are generally centered on a flow network formulation of the original problem. Replication in logic partitioning requires replication of the pins between the replicated cell (vertex) and its input nets thus, these input nets become cut after replication of that vertex.

One of the first iterative improvement based replication schemes is proposed by Kring and Newton [40]. They provide an extended version of the FM algorithm to handle replication in two-way partitioned networks by introducing new definitions for cell states and moves. There are two different move selection techniques introduced in their work: cleaning of unnecessary replications immediately even if there are higher gain moves and, the prohibition of replications whose gains are under a certain threshold. Kuznar et al. [8] introduced the concept of functional
CHAPTER 3. RELATED WORK

replication for partitioning a specific FPGA (Field Programmable Gate Array) library, on Xilinx-based devices. Their approach is similar to [40], as they use an FM heuristic by extending it to handle replication. The difference of their work lies in the definition of a cell, which stands for configurable logic blocks rather than logic gates as opposed to other approaches which use FM for VLSI circuit partitioning. In the replication schemes for logic partitioning, when a cell is replicated, all of its input pins must be replicated too. However, in functional replication, some input pins need not to be replicated since they may not be required to generate the replicated cell’s required outputs. In this way, they are able to remove some of the nets connected to input pins of the replicated cell which results in better cutsize values. Hwang and El Gamal [33, 34] formulate the min-cut replication problem and give the optimum solution for finding the min-cut replication sets of a $K$-way partitioned directed graph. Their approach uses a max-flow algorithm to find the replication set of a part that minimizes the cut. They show that this approach can independently be applied to find the replication sets of all parts for a given partition. The drawback of their algorithm is that the size of the replication sets are not guaranteed to be minimum; this may lead to unnecessary replications, and therefore, to the parts’ violation of their size limits. To solve the size constrained min-cut replication problem, they first apply the optimal min-cut replication algorithm on each part, and then, if any part violates its size constraint after the replication, a modified FM heuristic is used on those to find a feasible solution. Their solution for hypergraphs is straightforward, which is to replace each net with a directed tree and to use the same algorithm. However, this does not guarantee the minimum cutsize as opposed to partitioning graphs. Liu et al. [45] present an optimal algorithm for the two-way partitioning of graphs with replication and without size constraints. Their formulation of the problem requires a pair of source and sink nodes, rather than an initial partition as in [34]. They use the linear programming method for constructing the replication graph of the original graph, and then, a maximum flow algorithm on this replication graph to find the optimum replication schema. In the case of hypergraphs, they give a heuristic to construct the replication graph and extend FM to a directed FM to partition the replication graph (DFRG) for
minimizing the replication cut cost and satisfying the size constraints. Both approaches, [34] and [45], use a directed version of FM, where a part is chosen to be the source part and only the nets that have a source in the source part and a sink in the other part are considered to be in the cut. [34] uses it on the original graph while [45] uses it on the replication graph. Yang and Wong [63] propose algorithms for finding optimal solutions to the min-area min-cut replication problem for directed graphs and hypergraphs. In their work, they use different flow network models to find min-cut replication sets with minimum sizes. The graph flow network model they use is the same model used in [34]. They propose a new hypergraph flow network model which correctly models the hypergraph. In this approach, their algorithm searches the components in reverse order of [34] which leads to a smaller flow network model. Thus, their algorithm can find smaller cut sizes in a shorter amount of time. Detailed discussion and comparison of replication techniques in circuit partitioning can be found in [26].

In parallel information retrieval systems, the index is partitioned across several machines to be able to process very large text collections efficiently. There are typically two types of index partitioning schemes [57, 35, 12, 49, 51] in distributed IR: document partitioning and term partitioning. Replication is a widely used technique in parallel IR systems to improve query throughput and fault tolerance whatever the partitioning scheme is. Tomasic and Garcia-Molina [58] compare different index distributions in their work and conclude replication is necessary for improving query throughput. Lu and McKinley [47] compare the partial replication and caching to improve the IR system performance and conclude that although caching is simpler and faster, partial replication has the capability of exploiting locality of different queries that require similar answers. They extend their work in [48] by including a study of query locality and a replica selection function. In the distributed IR system of Google [5], the entire system is replicated in order to improve query throughput. Cacheda et al. [11] compare a replicated IR system with a distributed and clustered IR system. Moffat et. al [50] proposes selective inverted list replication to improve the load balancing in a pipelined and term-distributed IR system. Their selective replication approach replicates the inverted lists of high workload terms.
The replication in database systems is used for increasing availability and improving performance of the whole system [6, 7, 29, 61]. The replication is achieved by replicating records in the database across multiple sites. There is a compromise between the consistency of the replicas and the performance of the distributed database system. By replicating a record, the performance of the read operations is increased. However, this performance improvement is bounded by the write operations on the replicated records where serializability and consistency become critical issues as the number of replicas increases. There are mainly two techniques to solve the problem of consistency among multiple replicas. The first one is eager (pessimistic) replication [20, 37] where the write operation on a record is immediately propagated across other replicas and thus, there is one single version of a replica. The other technique is lazy (optimistic) replication [42, 56] where the updates are not propagated immediately and the different versions of the replicas may diverge. Another replication scheme is adaptive replication [62] where the replication of a record may change with respect to read and write patterns on that record. Replication in spatial databases [30, 53] is rather a new and unexplored topic although there are a few studies that explores replication of spatial data [52, 27].
Chapter 4

Replicated Hypergraph Partitioning

The multilevel framework discussed in Chapter 2.3 is enhanced to solve replicated HP problem. Specifically, the replication is achieved in the uncoarsening phase of the multilevel scheme by extending the FM heuristic that is used as the refinement algorithm. We call this extended heuristic replicated FM (rFM). Our proposed algorithm supports move, replication and unreplication operations on the vertices and it strives for minimizing the cutsize while maintaining balance as conventional FM-based algorithms. The multilevel replicated HP algorithm is used in a recursive bipartitioning framework (Section 2.3) to obtain a $K$-way replicated hypergraph partitioning. In this framework, after each bipartitioning, two sub-hypergraphs are constructed from the obtained partitions and these sub-hypergraphs are used for further bipartitions. In these sub-hypergraphs, the replicated vertices are considered as “real” vertices for the new hypergraphs and necessary pins are added. Thus, the coarsening and the initial partitioning phases of the multilevel scheme can be used as-is without being affected by the replication. After achieving a $K$-way partition by recursive bipartitioning, in order to compute the cutsize we have to decide which instances of the vertices will be used for each net. This decision affects the cutsize hence, must be done carefully.
4.1 Replicated FM (rFM)

4.1.1 Definitions

In a two-way replicated partition $\Pi^R = \{V_A, V_B\}$, a vertex can belong to $V_A$, $V_B$, or to both of them if it is replicated, and hence, it can be in one of three states $A$, $B$, $AB$.

$$\text{State}(v_i) = \begin{cases} A & \text{if } v_i \in V_A, \\ B & \text{if } v_i \in V_B, \\ AB & \text{if } v_i \in V_A \text{ and } v_i \in V_B. \end{cases}$$

We use the letters $A$ and $B$ to denote the parts of a bipartition. Each instance of a replicated vertex is referred to as replica.

The number of non-replicated vertices that are connected by $n_j$ in $A$ and $B$ are denoted as $\sigma_A(n_j)$ and $\sigma_B(n_j)$ respectively. Similarly, the number of replicated vertices that are connected by $n_j$ is represented by $\sigma_{AB}(n_j)$. In the examples, we use the notation $\sigma(n_j) = (\sigma_A(n_j) : \sigma_B(n_j) : \sigma_{AB}(n_j))$ to denote the pin distribution of $n_j$. Note that $|\text{Pins}(n_j)| = \sigma_A(n_j) + \sigma_B(n_j) + \sigma_{AB}(n_j)$. A net $n_j$ is said to be cut if $\sigma_A(n_j) > 0$ and $\sigma_B(n_j) > 0$. The cut-state of a net shows whether the net is cut.

There are 3 operations in rFM: move, replication and unreplication. The move and replication operations are defined for the non-replicated vertices, whereas the unreplication operation is defined for the replicated vertices. Therefore, a non-replicated vertex has move and replication gains whereas a replicated vertex has unreplication gains. Unreplication operation requires two different gain values to be maintained since there are two instances of a replicated vertex in $A$ and $B$, and each of them may have different unreplication gain. The gain definitions are as follows:

- Move gain, $g_m(v_i)$, is defined as the change in the cutsize if the vertex $v_i$ were to be moved to the other part. The move gain of $v_i$ is equal to the difference between the sum of the costs of the nets saved from the cut and
the sum of the costs of the internal nets that are brought to the cut. A simple example of move operation is seen in Figs. 4.1a and 4.1b. Moving $v_1$ from part $A$ to $B$ brings net $n_1$ into the cut while saving net $n_2$ from the cut. Hence, $g_m(v_1) = c(n_2) - c(n_1)$. After the move operation, $v_1$ is locked. The locked vertices in the examples are illustrated by gray color.

- **Replication gain**, $g_r(v_i)$, is defined as the change in the cutsize if the vertex $v_i$ were to be replicated to the other part. The replication gain of $v_i$ is equal to the sum of the costs of the nets saved from the cut. When a vertex is replicated, it cannot bring any internal net to the cut and thus, cannot increase the cutsize. This forms the basic difference between the move and the replication operation. Consequently, for any vertex $v_i$, we have $g_r(v_i) \geq 0$ and $g_r(v_i) \geq g_m(v_i)$. Figs. 4.1a and 4.1c show the replication of $v_1$ from part $A$ to $B$. The replication of $v_1$ saves net $n_2$ from the cut as the move of $v_1$ does, however, net $n_1$ still remains as an internal net, as opposed to the move operation on the same vertex. Hence, $g_r(v_1) = c(n_2)$. In the
examples, if a net is internal and connects a replicated vertex, we illustrate this by putting a pin to the replica that is in the part of the internal net and omit the pin to other replica. On the contrary, if an external net connects a replicated vertex, a replica of the replicated vertex is randomly chosen to include a pin to that external net.

![Diagram of unreplication process](image)

(a) Initial hypergraph before unreplication.

(b) After unreplicating the replica of $v_1$ from part $A$.

(c) After unreplicating the replica of $v_1$ from part $B$.

Figure 4.2: Unreplication of a replicated vertex.

- **Unreplication gain**, $g_{u,A}(v_i)$ or $g_{u,B}(v_i)$, is defined as the change in the cutsize if a replica of the replicated vertex $v_i$ were to be unreplicated from its current part. Since unreplication of a replica cannot improve the cutsize, the maximum unreplication gain of a replica is zero. Thus, for any replicated vertex $v_i$, $g_{u,A}(v_i) \leq 0$ and $g_{u,B}(v_i) \leq 0$. A replica that has an unreplication gain of zero means that this replica is unnecessary and its deletion will not change the cutsize. Furthermore, unreplication of a replica can be “harmful” by bringing internal nets to the cut. This means that the replica is necessary in which case it will have a negative unreplication gain. Fig. 4.2 shows the unreplication of a necessary and an unnecessary replica. Initially,
there are two replicas of $v_1$ in the sample hypergraph in Fig. 4.2a. The replica in part $A$ is necessary, and its deletion causes the internal net $n_1$ to be cut, as seen in Fig. 4.2b. On the other hand, the replica in part $B$ is unnecessary, and its deletion does not change the set of nets in the cut, as seen in Fig. 4.2c. Hence, $g_{u,A}(v_1) = -c(n_1)$ and $g_{u,B}(v_1) = 0$.

4.1.2 Overall rFM Algorithm

Replicated FM performs predetermined number of passes over all vertices where each pass comprises of a sequence of operations. Each iteration of a pass starts with selection of a vertex and an operation (move, replication or unreplication) to be performed on that vertex. The selected operation must not violate the size constraints on the weights of the parts. After the selected operation is applied on the vertex, the vertex is locked and the gain values of its unlocked neighbors and the pin distributions of its nets are updated. The size constraints need to be updated if the performed operation is replication or unreplication since the total vertex weight changes. A pass stops when there are no more valid operations. At the end of a pass, a rollback procedure is applied to the point where the partition with the minimum cutsize is seen and then, all vertices are unlocked for further passes. These basic steps are shown in Algorithm 1.

<table>
<thead>
<tr>
<th>Algorithm 1: Basic Steps of RFM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi^R = {V_A, V_B}$</td>
</tr>
<tr>
<td>1 while there are passes to perform do</td>
</tr>
<tr>
<td>2 while there is any valid operation do</td>
</tr>
<tr>
<td>3 \quad \quad (v, op) ← Select the vertex and the operation to perform on it with respect to the selection criteria.</td>
</tr>
<tr>
<td>4 \quad \quad Perform op on $v$, store the change in the cutsize and lock $v$.</td>
</tr>
<tr>
<td>5 \quad \quad Update gains of unlocked neighbors of $v$ and pin distributions of $\text{Nets}(v)$.</td>
</tr>
<tr>
<td>6 if op is replication or unreplication then</td>
</tr>
<tr>
<td>7 \quad \quad Update size constraints of parts.</td>
</tr>
<tr>
<td>8 \quad \quad Unlock all vertices and rollback to the point where the minimum cutsize is seen.</td>
</tr>
</tbody>
</table>
4.1.2.1 Operation Selection

We use a priority based approach for selecting the current operation and disallow some operations that do not satisfy certain conditions. The selection strategy is based on principles such as minimizing the number of unnecessary replicas, limiting replication amount and improving balance.

The highest priority is given to the unreplication operation with a gain value of zero to get rid of unnecessary replica(s) immediately. If a replica has a negative unreplication gain, it means this replica is necessary and its deletion will cause an increase in the cutsize. We do not perform unreplication operations with negative gains and therefore, we do not allow necessary replicas to be deleted.

If there is no replica with an unreplication gain of zero, we make a choice between move and replication by selecting the operation with the higher gain. In the case where the gains of the examined move and replication operations are equal, the move operation is given a higher priority in order to not to consume the given replication allowance. The replication with a gain value of zero is disallowed simply because such operations will produce unnecessary replicas; however, the zero gain moves that improve the balance are performed. Since for any vertex \( v_i \), \( g_r(v_i) \geq g_m(v_i) \), in a single pass, the number of replication operations tends to outweigh the number of move operations. This issue can be addressed by the gradient methodology, which is discussed in the following subsections.

4.1.2.2 Gradient Methodology

The gradient methodology is proposed and used for the FM heuristics that is capable of replication in [26, 46] to obtain partitions with better cutsize. The basic idea of the gradient methodology is to introduce the replication in the later iterations of a pass, especially when the improvement achieved in the cutsize by performing only move operations drops below a certain threshold. As mentioned in [26], early replication can have a negative effect on the final partition by limiting the algorithm’s ability to change the current partition. Furthermore, by using the
replication in the later iterations, the algorithm can proceed by using the local
minima reached by the move operations. In rFM, we adopt and modify this
methodology by using move and unreplication operations till the improvement
in the cutsize drops below a certain threshold and then, we allow replication
operations.

4.1.2.3 Early-Exit

We adopt the early-exit scheme for rFM which is a well-known technique to
improve the running time performance of the FM-based heuristics. In the early-
exit scheme, if there are no improvements in the cutsize for a predetermined
number of iterations, the algorithm stops because it is unlikely to further improve
the partition.

4.1.2.4 Locking

In conventional move-based FM algorithms, after moving a vertex, it is locked
to avoid thrashing. Similarly, in rFM, we also lock the operated vertex after
performing a move operation. Furthermore, after performing a replication op-
eration on \( v_i \in V_A \), both replicas of \( v_i \) are locked. The replica of \( v_i \) in part \( A \)
is locked since the unreplication of this replica would leave the replica in part
\( B \) alone where performing these two operations (replicating \( v_i \in V_A \) and then
unreplication of its replica in part \( A \)) simply becomes equivalent to performing
a move operation on \( v_i \). The unreplication gain of the replica in part \( B \) will be
negative after the replication of \( v_i \) since only positive gain valued replications
are allowed which means this replica is necessary and thus, it is locked. Finally,
after unreplication of a replicated vertex, the remaining replica, which is now a
non-replicated vertex, is locked. That is because the move or replication of this
non-replicated vertex will not improve the cutsize.
4.1.2.5 Data Structures

We maintain 6 priority queues keyed according to the gain values of the vertices with respect to operations. For efficiency purposes, the priority queues are implemented as binary heaps. We do not use buckets for storing gain values since the buckets do not perform well when the variation between net costs is high. There are two heaps for the move gains (heapMA and heapMB), two heaps for the replication gains (heapRA and heapRB), and two heaps for the unreplication gains (heapUA and heapUB). A non-replicated vertex has its move and replication gains stored in two heaps (either in heapMA and heapRA or heapMB and heapRB). Similarly, the two replicas of a replicated vertex have their unreplication gains stored in two heaps (in heapUA and heapUB).

In the selection of the vertex and the operation to be performed on that vertex, the root nodes of all heaps are retrieved and the selection is done according to the criteria mentioned above. After the selection is done, we perform an extract-max operation on the heap of the selected vertex and a delete operation on another heap since the selected vertex possesses another gain value. The deletion operation for the other gain value of the selected vertex is required since the vertex is locked and no further operation should be available for this vertex throughout the current pass. For instance, if an extract-max operation is performed on heapMA, it is required to perform a delete operation on heapRA or vice versa. Similarly, if an extract-max operation is performed on heapUA, we need to perform a delete operation on heapUB. Performing an operation may cause gain updates on the neighbors of the operated vertex. A gain update for any vertex may require increase-key or decrease-key operation on the heaps of this vertex belong to.

4.1.3 Initial Gain Computation and Gain Update Algorithms

In this section, we describe the initial gain computation and gain update algorithms in detail. The gain values of the vertices need to be computed at the
beginning of each pass of rFM. After each operation, gain updates may be re-
quired for the neighbors of the operated vertex. For each neighbor of the operated
vertex, there are at most two gain updates since any vertex possesses two gain
values. When compared to the conventional FM-based algorithms in the VLSI
literature, rFM has more gain updates and hence, more heap operations. The
examples in this section respect to the basics of the operation selection criteria
mentioned in Section 4.1.2. For the sake of simplicity, we assume each net has
unit cost.

A net \( n_j \) is said to be critical if an operation on this net would change its
cut-state. A net can be critical to a part with respect to its pin distribution.
Each type of operation imposes different pin distributions for the criticality of
\( n_j \). In the conventional move-based FM algorithm, the move gains are updated
whenever the criticality of a net changes. It is interesting to note that the same
applies for the update of the replication and the unreplication gains where the
criticality of the nets for these two operations is a subset of the criticality of the
nets for the move operation. Clearly, the criticality of a net for the move and
the replication operation requires at least two non-replicated vertices to exist for
that net \( (\sigma_A(n_j) + \sigma_B(n_j) > 1) \), since the nets that have a single non-replicated
pin cannot be critical for the move and replication operations \( (\sigma_A(n_j) = 1 \) or
\( \sigma_B(n_j) = 1) \). For the move operation,

\[
\begin{align*}
n_j \text{ is move-critical to part } A & \text{ if } \sigma_A(n_j) = 1 \text{ or } \sigma_B(n_j) = 0, \\
n_j \text{ is move-critical to part } B & \text{ if } \sigma_B(n_j) = 1 \text{ or } \sigma_A(n_j) = 0.
\end{align*}
\]

(4.1) (4.2)

The criticality of the internal nets in the move operation is not valid for the
replication operation since the replication of a vertex connected to an internal
net cannot change the cut-state of that net. However, as in the move operation,
the external nets having only one non-replicated pin in a part are critical to that
part. Thus, for the replication operation,

\[
\begin{align*}
n_j \text{ is replication-critical to part } A & \text{ if } \sigma_A(n_j) = 1, \\
n_j \text{ is replication-critical to part } B & \text{ if } \sigma_B(n_j) = 1.
\end{align*}
\]

(4.3) (4.4)

For a net to be critical for the unreplication operation, it must have at least one
replicated vertex ($\sigma_{AB}(n_j) > 0$). As mentioned in Section 4.1.2, the unreplication operation can only change the cut-state of the internal nets. Thus, criticality of a net for the unreplication operation requires it to be internal to a part. Similar to the criticality of the internal nets in the move operation, unreplication operation imposes the same conditions on the criticality of nets. There is an exception for the unreplication operation where a net can be internal but still may not be critical. This exception occurs if the net has pins only to the replicated vertices meaning that the net can be internal to any part, i.e., $\sigma_A(n_j) = \sigma_B(n_j) = 0$ and $\sigma_{AB}(n_j) > 0$. In this condition, unreplication of any replica connected to this net will not make it external. Such nets are also not critical for the move operation since they do not have any non-replicated pins to operate on. Thus, the criticality conditions of the nets for the unreplication operation are still a subset of the criticality conditions of the nets for the move operation. In Fig. 4.3, the net $n_j$ has three replicated vertices $v_r, v_s, v_t$ and no non-replicated vertices. In this case, $n_j$ can be internal to any part as seen in Figs. 4.3a and 4.3b. Unreplication of a replica connected by $n_j$ does not make it external since the number of non-replicated vertices connected by $n_j$ will be equal to one after unreplication and, single pin nets are clearly internal nets as explained. Fig. 4.4 shows various unreplication operations on the bipartition in Fig. 4.3. As seen in Fig. 4.4a, unreplication of the replica of $v_s$ from part $B$ does not make $n_j$ external. Similarly, unreplication of the replica of $v_s$ from part $A$ does not make $n_j$ external in Fig. 4.4b. However, two unreplication operations on $n_j$, one from part $A$ and one from part $B$, will make $n_j$ external since $\sigma_A(n_j) > 0$ and $\sigma_B(n_j) > 0$ after these operations. This condition is illustrated in Fig. 4.4c where the replica of $v_r$
in part $A$ and the replica of $v_s$ in part $B$ are unreplicated. Consequently, for the unreplication operation,

\begin{align}
    n_j \text{ is unreplication-critical to part } A & \text{ if } \sigma_B(n_j) = 0 \text{ and } \sigma_A(n_j) > 0, \\
    n_j \text{ is unreplication-critical to part } B & \text{ if } \sigma_A(n_j) = 0 \text{ and } \sigma_B(n_j) > 0.
\end{align}

Figure 4.4: Various unreplication operations on the replicas of a net that has no non-replicated vertices.

### 4.1.3.1 Initial gain computation

The initial gain computation given in Algorithm 2 consists of two main loops. The first loop resets the initial gain values by traversing vertices (lines 1–7) and the second loop completes the initialization of gains by traversing pins of the nets (lines 8–20). While resetting gain values, all nets are considered to be internal for the move gains, gainless for the replication gains and external for the unreplication gains. Then, in the completion of the gain values, the move and replication gains are modified for the external and/or critical nets whereas the unreplication gains are modified for the internal (or critical) nets.

The move and replication gains of the non-replicated vertices are initially set to their minimum possible values. If a net $n_j$ is external, its pins’ move and replication gains may need to be updated. If this external net is move- or replication-critical to a part (see Equations 4.1, 4.2, 4.3 and 4.4), the move and
Algorithm 2: Initial move, replication and unreplication gain computation

Input: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi^R = \{V_A, V_B\}$

1. foreach $v_i \in \mathcal{V}$ do
   2. if $\text{State}(v_i) \neq AB$ then
      3. $g_m(v_i) \leftarrow -c(\text{Nets}(v_i))$
      4. $g_r(v_i) \leftarrow 0$
   5. else
      6. $g_{u,A}(v_i) \leftarrow 0$
      7. $g_{u,B}(v_i) \leftarrow 0$

8. foreach $n_j \in \mathcal{N}$ do
   9. foreach $v_i \in \text{Pins}(n_j)$ do
      10. if $\text{State}(v_i) \neq AB$ and $n_j$ is external then
          11. if $(\sigma_A(n_j) = 1$ and $\text{State}(v_i) = A)$ or $(\sigma_B(n_j) = 1$ and $\text{State}(v_i) = B)$ then $\triangleright$ $n_j$ is critical to part $A$ or $B$
              12. $g_m(v_i) \leftarrow g_m(v_i) + 2 \cdot c(n_j)$
              13. $g_r(v_i) \leftarrow g_r(v_i) + c(n_j)$
          14. else
              15. $g_m(v_i) \leftarrow g_m(v_i) + c(n_j)$
      16. else if $\text{State}(v_i) = AB$ and $n_j$ is internal then
          17. if $\sigma_A(n_j) > 0$ and $\sigma_B(n_j) = 0$ then $\triangleright$ $n_j$ is critical to part $A$
              18. $g_{u,A}(v_i) \leftarrow g_{u,A}(v_i) - c(n_j)$
          19. else if $\sigma_B(n_j) > 0$ and $\sigma_A(n_j) = 0$ then $\triangleright$ $n_j$ is critical to part $B$
              20. $g_{u,B}(v_i) \leftarrow g_{u,B}(v_i) - c(n_j)$
replication gains of the vertex which is in the part that \( n_j \) is critical to must be updated since it can be saved from the cut with either move or replication. Its move gain is increased twice \( c(n_j) \) since initially all nets are considered to be internal for the move operation and its replication gain is increased by \( c(n_j) \). On the other hand, if this external net is not move- or replication-critical, the move gain of its pins are increased by \( c(n_j) \). This can be seen as a simple correction for considering all nets as internal while resetting move gain values.

In contrast to move and replication gains, unreplication gains are initially set to their maximum possible values. If a net \( n_j \) is unreplication-critical and thus internal (see Equations 4.5 and 4.6), the unreplication gains of its replicated pins may need to be updated. The unreplication gains of the replicas that are in the same part with this internal net need to be decremented by \( c(n_j) \) if \( n_j \) has at least one non-replicated vertex in the same part.

Fig. 4.5 shows a sample bipartition, the pin distributions of the nets and the
gain values of the vertices in this hypergraph after Algorithm 2 is run. As seen in the figure, the nets \( n_1, n_3 \) and \( n_5 \) are move-critical to part \( A \), whereas \( n_2, n_4, n_6 \) and \( n_7 \) are move-critical to part \( B \). The internal nets which are critical for the move operation cannot be critical for the replication operation. If we omit such nets from the set of the critical nets for the move operation, we get the critical nets for the replication operation. Thus, \( n_5 \) is replication-critical to part \( A \) while \( n_4 \) and \( n_6 \) are replication-critical to part \( B \). Only internal nets that connects at least one replicated vertex can be critical for the unreplication operation. In this case, \( n_1 \) is unreplication-critical to part \( A \) and \( n_2 \) is unreplication-critical to part \( B \). The nets \( n_4, n_5 \) and \( n_6 \) are in the cut thus, the cutsize of the bipartition in Fig. 4.5 is 3.

4.1.3.2 Gain updates after move operation

Algorithm 3 shows the procedure of gain updates after moving the given vertex \( v^* \) from part \( A \) to \( B \). The algorithm includes updating fields of \( v^* \), the pin distributions of \( \text{Nets}(v^*) \) and the gain values of neighbors of \( v^* \). The necessary field updates on \( v^* \) are performed by updating the state and locked fields of \( v^* \) to reflect the move operation. The pin distribution of each net \( n_j \in \text{Nets}(v^*) \) needs to be updated by decrementing \( \sigma_A(n_j) \) by 1 and incrementing \( \sigma_B(n_j) \) by 1. When the pin distribution of \( n_j \) changes, its criticality may change with respect to operation type for part \( A \) or \( B \). The change in the criticality of \( n_j \) may require various gain updates on the unlocked pins of this net.

After decrementing the number of pins of \( n_j \) in \( A \), we check the value of \( \sigma_A(n_j) \) to see if the criticality of \( n_j \) has changed. If \( \sigma_A(n_j) = 0 \), \( n_j \) becomes internal to part \( B \) by becoming move- and unreplication-critical to this part (see Equations 4.2 and 4.6). In this case, the move and the unreplication gains of the unlocked vertices and replicas connected to \( n_j \) in \( B \) need to be decremented by \( c(n_j) \). If \( \sigma_A(n_j) = 1 \), \( n_j \) becomes move- and replication-critical to part \( A \) (see Equations 4.1 and 4.3). The only vertex of \( n_j \) in \( A \) can now save \( n_j \) from the cut and thus, the move and replication gains of this vertex must be incremented by \( c(n_j) \) if it is unlocked.
Algorithm 3: Gain updates after moving $v^*$ from part $A$ to $B$

Input: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi^R = \{V_A, V_B\}, v^* \in V_A$

1. $\text{State}(v^*) \leftarrow B$
2. Lock $v^*$
3. foreach $n_j \in \text{Nets}(v^*)$ do
   4. $\sigma_A(n_j) \leftarrow \sigma_A(n_j) - 1$
   5. if $\sigma_A(n_j) = 0$ then /* $n_j$ becomes critical to part $B$ */
      6. foreach unlocked $v_i \in \text{Pins}(n_j)$ do
          7. if $\text{State}(v_i) = B$ then
              8. $g_m(v_i) \leftarrow g_m(v_i) - c(n_j)$
          else if $\text{State}(v_i) = AB$ then
              9. $g_{u,A}(v_i) \leftarrow g_{u,A}(v_i) - c(n_j)$
      else if $\sigma_A(n_j) = 1$ then /* $n_j$ becomes critical to part $A$ */
      10. foreach unlocked $v_i \in \text{Pins}(n_j)$ do
          11. if $\text{State}(v_i) = A$ then
              12. $g_m(v_i) \leftarrow g_m(v_i) + c(n_j)$
              13. $g_r(v_i) \leftarrow g_r(v_i) + c(n_j)$
          else if $\sigma_B(n_j) = 2$ then /* $n_j$ was critical to part $B$ */
      14. $\sigma_B(n_j) \leftarrow \sigma_B(n_j) + 1$
      15. if $\sigma_B(n_j) = 1$ then /* $n_j$ was critical to part $A$ */
      16. foreach unlocked $v_i \in \text{Pins}(n_j)$ do
          17. if $\text{State}(v_i) = A$ then
              18. $g_m(v_i) \leftarrow g_m(v_i) + c(n_j)$
          else if $\text{State}(v_i) = AB$ then
              19. $g_{u,A}(v_i) \leftarrow g_{u,A}(v_i) + c(n_j)$
      20. else if $\sigma_B(n_j) = 2$ then /* $n_j$ was critical to part $B$ */
      21. foreach unlocked $v_i \in \text{Pins}(n_j)$ do
          22. if $\text{State}(v_i) = B$ then
              23. $g_m(v_i) \leftarrow g_m(v_i) - c(n_j)$
              24. $g_r(v_i) \leftarrow g_r(v_i) - c(n_j)$
After incrementing the number of pins of \( n_j \) in \( B \), we check the value of \( \sigma_B(n_j) \) to see if the criticality of \( n_j \) has changed. If \( \sigma_B(n_j) = 1 \), it means \( n_j \) was internal and move- and unreplication-critical to part \( A \) (see Equations 4.1 and 4.5). Now because \( n_j \) becomes external, the move of the vertices or the unreplication of the replicas connected to \( n_j \) in \( A \) will not make it external. Thus, the move and the unreplication gains of the unlocked vertices and the replicas in \( B \) need to be incremented by \( c(n_j) \). Finally, if \( \sigma_B(n_j) = 2 \), it means \( n_j \) was move- and replication-critical to part \( B \) (see Equations 4.2 and 4.4). Before the move of \( v^* \), \( n_j \) had only one vertex in \( B \) which can save \( n_j \) from the cut. However, after moving \( v^* \) to \( B \), \( n_j \) now has two vertices in \( B \) and it cannot be saved from the cut anymore. Hence, the move and replication gains of unlocked pin of \( n_j \) in \( B \) must be decremented by \( c(n_j) \).

In Fig. 4.5, when we consider the selection criteria, since there are no replicas with a gain value of zero, we are to select move or replication operation. The highest move gain value is equal to the highest replication gain value which is 1. In such a condition where a tie-break occurs, we select the move operation which...
in this case is the move of $v_4$. Fig. 4.6 shows the bipartition after moving and locking $v_4$. After updating the pin distributions of $Nets(v_4) = \{n_5, n_6\}$, the gain values of the neighbors of $v_4$ may need to be updated. The pin distribution of $n_5$ becomes $\sigma_A(n_5) = 0$ and $\sigma_B(n_5) = 3$. The net $n_5$ becomes critical to part $B$ and thus, the move gains of the pins of $n_5$ in $B$, $v_6$ and $v_7$, are decreased by one. The pin distribution of $n_6$ becomes $\sigma_A(n_6) = 1$ and $\sigma_B(n_6) = 2$. Since $\sigma_A(n_6) = 1$, $n_6$ becomes critical to part $A$ which means the move and replication gains of $v_5$ need to be incremented by one. Finally, $n_6$ was critical to part $B$ before the movement of $v_4$ meaning that the move and replication gains of $v_7$ need to be decremented by one. When the gain updates are completed, the cutsize of the bipartition becomes 2. The pin distributions and the gain values after running Algorithm 3 are shown in the table in Fig. 4.6.

4.1.3.3 Gain updates after replication operation

Algorithm 4 shows the procedure of gain updates after replicating the given vertex $v^*$ from $A$ to $B$. The procedure starts with changing the state of $v^*$ to replicated $(AB)$ and locking both replicas of $v^*$. Then, for each net $n_j$ connecting $v^*$, the pin distributions of $n_j$ are updated and checked for their criticality conditions whether they changed or not. Since $v^*$ was in $A$ before replication, $\sigma_A(n_j)$ is decremented by 1 and, $\sigma_{AB}(n_j)$ is incremented by 1 since $v^*$ is replicated now. The replication of $v^*$ from $A$ does not change the $\sigma_B(n_j)$ values of the nets that connect $v^*$, thus the criticality conditions that include $\sigma_B(n_j)$ need not to be checked.

After decremnting $\sigma_A(n_j)$ for each $n_j \in Nets(v^*)$, we check the pin distribution of $n_j$ regarding $\sigma_A(n_j)$ for the criticality conditions. If $\sigma_A(n_j) = 0$, $n_j$ becomes move- and unreplication-critical for part $B$ (see Equations 4.2 and 4.6). In this condition, the move gains of the unlocked vertices and the unreplication gains of the unlocked replicas connected to $n_j$ need to be decremented by $c(n_j)$ since $n_j$ is internal now and the move of any vertex or the unreplication of any replica would bring it to cut. There are some exceptional cases for the replication and the unreplication operations since the value of $\sigma_B(n_j)$ does not change. One
Algorithm 4: Gain updates after replicating $v^*$ from part $A$ to $B$

Input: $\mathcal{H} = (\mathcal{V}, \mathcal{N}), \Pi^R = \{V_A, V_B\}, v^* \in V_A$

1. $\text{State}(v^*) \leftarrow AB$
2. Lock $v^*$
3. foreach $n_j \in \text{Nets}(v^*)$ do
   4. $\sigma_A(n_j) \leftarrow \sigma_A(n_j) - 1$
   5. $\sigma_{AB}(n_j) \leftarrow \sigma_{AB}(n_j) + 1$
   6. if $\sigma_A(n_j) = 0$ then \(\triangleright n_j \text{ becomes critical to part } B\)
      7. foreach unlocked $v_i \in \text{Pins}(n_j)$ do
         8. if $\text{State}(v_i) = B$ then
            9. $g_m(v_i) \leftarrow g_m(v_i) - c(n_j)$
            10. if $\sigma_B(n_j) = 1$ then
                11. $g_r(v_i) \leftarrow g_r(v_i) - c(n_j)$
            else if $\text{State}(v_i) = AB$ then
                12. if $\sigma_B(n_j) = 0$ then
                    13. $g_{u,A}(v_i) \leftarrow g_{u,A}(v_i) + c(n_j)$
                else if $\sigma_B(n_j) > 0$ then
                    14. $g_{u,B}(v_i) \leftarrow g_{u,B}(v_i) - c(n_j)$
            else if $\sigma_A(n_j) = 1$ then \(\triangleright n_j \text{ becomes critical to part } A\)
                15. foreach unlocked $v_i \in \text{Pins}(n_j)$ do
                    16. if $\text{State}(v_i) = A$ then
                        17. $g_m(v_i) \leftarrow g_m(v_i) + c(n_j)$
                        18. if $\sigma_B(n_j) > 0$ then
                            19. $g_r(v_i) \leftarrow g_r(v_i) + c(n_j)$
of these cases occurs for the replication operation if \( \sigma_B(n_j) = 1 \) where the replication of the only vertex connected to \( n_j \) in \( B \) will not save \( n_j \) from cut anymore and thus, its replication gain must be decremented by \( c(n_j) \). The other case is for the unreplication operation and occurs if \( \sigma_B(n_j) = 0 \) which means there are no non-replicated vertices connected to \( n_j \). In this case, unreplication of the replicas in \( A \) were bringing \( n_j \) to the cut before the replication of \( v^* \). However, after replication of \( v^* \), unreplication of these replicas will not bring \( n_j \) to the cut and thus, their unreplication gains must be incremented by \( c(n_j) \).

If \( \sigma_A(n_j) = 1 \), \( n_j \) becomes move- and replication-critical to part \( A \) (see Equations 4.1 and 4.3). The move or the replication of the only non-replicated vertex \( v_i \) connected to \( n_j \) in \( A \) can now save \( n_j \) from the cut and thus, the move and replication gain of this vertex must be incremented by \( c(n_j) \). However, the replication gain of \( v_i \) needs to be incremented only if \( \sigma_B(n_j) > 0 \). That is because in the condition where \( \sigma_B(n_j) = 0 \) before the replication of \( v^* \), the replication of \( v_i \) will not change the cutsize and, after the replication of \( v^* \), the replication of \( v_i \) will still not change the cutsize of the partition. Thus, if \( \sigma_B(n_j) = 0 \), the replication gain of \( v_i \) need not be incremented.

After moving \( v_4 \), now we are to select another vertex to operate on in Fig. 4.6. There are two operations with the highest gain which are the replication of \( v_5 \) and the replication of \( v_6 \) and the gain values of these operations are 1. We select to replicate \( v_6 \). Fig. 4.7 shows the bipartition after replicating \( v_6 \) and locking both replicas of it. After the pin distributions of \( \text{Nets}(v_6) = \{n_2, n_4, n_5, n_7\} \) are updated, the criticalities of \( n_2, n_4 \) and \( n_7 \) change and the gain values of the vertices connected to these nets may need to be updated. For \( n_2 \), \( \sigma_B(n_2) = 0 \) which makes it critical to part \( A \) and since \( \sigma_A(n_2) = 0 \), it has no non-replicated vertices. Thus, the unreplication gain of the replica of \( v_1 \) in \( B \) is incremented by one. The pin distribution of \( n_4 \) for \( B \) becomes \( \sigma_B(n_4) = 0 \) making it critical to part \( A \). The move gains of the vertices connected to \( n_4 \) in \( A \), \( v_2 \) and \( v_3 \), need to be decremented by one, however, since \( \sigma_A(n_4) = 2 \), the replication gain values of these two vertices need not to be updated. Finally, \( n_7 \) becomes critical to part \( B \) since \( \sigma_B(n_7) = 1 \) and thus the move and replication gains of the only non-replicated vertex in \( B \) that is connected to \( n_7 \) may need to be updated. This
vertex is \( v_7 \) and its move gain is incremented by one, however, its replication gain is not incremented since \( \sigma_A(n_j) = 0 \). After the gain updates, the cutsize of the bipartition becomes 1. The gain values and the pin distributions after running Algorithm 4 are shown in Fig. 4.7.

4.1.3.4 Gain updates after unreplication operation

Algorithm 5 shows the procedure of gain updates after unreplication of the given replica \( v^* \) from \( A \). Firstly, the state of the replicated vertex \( v^* \) is changed to \( B \) and it is locked. The pin distributions of each net \( n_j \in \text{Nets}(v^*) \) are updated by incrementing \( \sigma_B(n_j) \) by 1 and decrementing \( \sigma_{AB}(n_j) \) by 1. Then, the criticality conditions of the nets connected to \( v^* \) are checked for the gain updates of the neighbors of \( v^* \). Since the value of \( \sigma_A(n_j) \) does not change, it is not necessary to check the criticality conditions that include \( \sigma_A(n_j) \).

After the value of \( \sigma_B(n_j) \) is incremented, the criticality of \( n_j \) must be checked.
Algorithm 5: Gain updates after unreplicating $v^*$ from part $A$

Input: $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi^R = \{V_A, V_B\}$; $v^* \in V_A$

1. $State(v^*) \leftarrow B$
2. Lock $v^*$

3. foreach $n_j \in Nets(v^*)$ do
   4. $\sigma_B(n_j) \leftarrow \sigma_B(n_j) + 1$
   5. $\sigma_{AB}(n_j) \leftarrow \sigma_{AB}(n_j) - 1$
   6. if $\sigma_B(n_j) = 1$ then ▷ $n_j$ was critical to part $A$
      7. foreach unlocked $v_i \in Pins(n_j)$ do
         8. if $State(v_i) = A$ then
            9. $g_m(v_i) \leftarrow g_m(v_i) + c(n_j)$
            10. if $\sigma_A(n_j) = 1$ then
                11. $g_r(v_i) \leftarrow g_r(v_i) + c(n_j)$
            else if $State(v_i) = AB$ then
                12. if $\sigma_A(n_j) = 0$ then
                    13. $g_{u,B}(v_i) \leftarrow g_{u,B}(v_i) - c(n_j)$
                else if $\sigma_A(n_j) > 0$ then
                    14. $g_{u,A}(v_i) \leftarrow g_{u,A}(v_i) + c(n_j)$
            else if $\sigma_B(n_j) = 2$ then ▷ $n_j$ was critical to part $B$
                15. foreach unlocked $v_i \in Pins(n_j)$ do
                   16. if $State(v_i) = B$ then
                       17. $g_m(v_i) \leftarrow g_m(v_i) - c(n_j)$
                       18. if $\sigma_A(n_j) > 0$ then
                           19. $g_r(v_i) \leftarrow g_r(v_i) - c(n_j)$
}


to see if there are any necessary gain updates for the neighbors of \( v^* \). If \( \sigma_B(n_j) = 1 \), it means \( n_j \) was move- and unreplication-critical to part \( A \) (see Equations 4.1 and 4.5). In this case, the move and the replication gains of the unlocked vertices and the unlocked replicas in \( A \) are increased by \( c(n_j) \) since \( n_j \) is no more an internal net. Similar to the gain updates after replication of a vertex, there are two exceptional cases. The first case occurs for the replication operation if \( \sigma_A(n_j) = 1 \) where the replication gain of the only vertex of \( n_j \) in \( A \) needs to be incremented since this vertex can save \( n_j \) from cut. The other case is for the unreplication operation and occurs when \( \sigma_A(n_j) = 0 \) meaning that \( n_j \) had no non-replicated vertex before the unreplication of \( v^* \) and unreplication of the replicas in \( B \) were not bringing \( n_j \) to the cut. However, after the unreplication of \( v^* \), the unreplication of the replicas in \( B \) will bring \( n_j \) to the cut and thus, the unreplication gains of these replicas must be decremented by \( c(n_j) \).

If \( \sigma_B(n_j) = 2 \), it means \( n_j \) was move- and replication-critical to part \( B \) (see Equations 4.2 and 4.4). In this case, \( n_j \) has two pins in \( B \) and one of them, \( v^* \), is already locked. The move and replication gains of the other vertex, \( v_i \), need to be decremented by \( c(n_j) \) since this vertex can no more save \( n_j \) from cut. However, if \( \sigma_A(n_j) = 0 \), it is not necessary to decrement the replication gain of \( v_i \). That is because the replication of \( v_i \) does not change the cutsize before or after the unreplication of \( v^* \).

In Fig. 4.7 after the replication of \( v_6 \), there exists an unnecessary replica in part \( B \) with an unreplication gain of zero. According to the selection criteria, the unreplication operation with a gain of zero has the highest priority. Thus, the selected operation is the unreplication of an unnecessary replica which is the unreplication of \( v_1 \) from \( B \). Fig. 4.8 shows the bipartition after the unreplication of \( v_1 \) from \( B \) and locking it. After the pin distributions of \( \text{Nets}(v_1) = \{n_1, n_2\} \) are updated, the gains of neighbors of \( v_1 \) may need to be updated. The pin distribution of \( n_1 \) for \( A \) is \( \sigma_A(n_1) = 3 \) and its criticality has not changed. On the other hand, the pin distribution of \( n_2 \) for \( A \) is \( \sigma_A(n_2) = 1 \) which means \( n_2 \) was critical to part \( A \). However, since none of the vertices connected to \( n_2 \) is unlocked, there is no need to perform gain updates for the pins of \( n_2 \). The unreplication of an unnecessary replica cannot change the cutsize thus, after the gain updates,
Figure 4.8: Gains and pin distributions after unreplicating $v_1$ from $V_B$

the cutsize is still 1. The gain values and the pin distributions after running Algorithm 5 are shown in Fig. 4.8.

4.1.3.5 Complexity Analysis of rFM

A single pass of rFM consists of initial gain computation, repeatedly selecting a vertex to operate on and gain updates of neighbors of the selected vertex after performing an operation on that vertex (see Algorithm 1). The total number of vertices is equal to the sum of the number of non-replicated vertices and the number of replicated vertices. This is because, in our implementation, when a vertex is replicated, the new replica of this vertex is not added to the data structure of the current hypergraph throughout the corresponding uncoarsening phase. For a replicated vertex connected to a net, there is only a single pin of that net for the replicated vertex. Thus, clearly, the number of vertices and the number of pins of the hypergraph at the beginning of a coarsening phase will be equal to the number of vertices and the number of pins of the hypergraph at the end of
the corresponding uncoarsening phase. Section 4.3 explains how the replicated vertices and their pins are handled in the construction of sub-hypergraphs for further bisections.

Let \( n \) and \( p \) be the number of vertices and the number of pins of a given bipartition \( \Pi^R = \{V_A, V_B\} \) on \( \mathcal{H} = (\mathcal{V}, \mathcal{N}) \) for a pass of rFM. Let \( r \) be the number of replicated vertices and \( s \) be the number of non-replicated vertices. Clearly, \( n = r + s \). The initial gain computation takes \( O(p) \) time since in Algorithm 2, the pins of each net are traversed. After the initial gain computation is completed, these gain values are stored in 6 heaps. For each heap, it is required to perform build-heap operations. The build-heap operations on heapMA and heapMB will take a total of \( O(s) \) time and similarly, the build-heap operations on heapRA and heapRB will take a total of \( O(s) \) time since the number of vertices on heapMA and heapMB or heapRA and heapRB is equal to number of non-replicated vertices, \( s \) (see Section 4.1.2 for the abbreviations of the heaps we use). The build-heap operation on heapUA will take \( O(r) \) time and similarly, the build-heap operation on heapUB will take \( O(r) \) time since each heap possesses \( r \) number of elements. Thus, the total time required for building heaps is equal to \( O(r + r + s + s) = O(2n) = O(n) \).

The selection procedure consists of checking maximum gain values in 6 heaps, which takes constant amount of time. After selecting the gain value from one of the heaps with respect to the selection criteria, we perform an extract-max operation on the selected heap and a delete operation on another heap for the other gain value of the selected vertex (Section 4.1.2). No matter the selected heap, the extract-max and delete operations on the heaps will be bounded by the number of total vertices since the maximum number of elements in a single heap can be at most \( n \). Thus, a single selection operation will take \( O(1 + 2 \log n) = O(\log n) \). In a single pass of rFM where all vertices are exhausted, we can make at most \( n \) selections. Consequently, the cost of selection in a single pass of rFM is equal to \( O(n \log n) \).

In the original move based FM algorithm [28], the gain updates take \( O(p) \) time since in a single pass, there are at most three update operations for the vertices
connected to \( n_j \) and, the move gain of a vertex connected to \( n_j \) is updated at most twice on these three update operations. This is due to the critical net definitions for the move operations and the locking schemes the FM algorithm uses [28]. In rFM, we use the same critical net definitions and locking schemes as the original FM algorithm. The critical net definitions of the newly introduced replication and unreplication operations are subset of the critical net definitions of the move operation (see Section 4.1.3). Thus, the complexity of gain updates in a single pass of rFM is the same as the FM heuristic. In rFM, since each vertex possesses two gains, in the worst case, both of these gain values may need to be updated, which doubles the number of gain updates for any vertex compared to FM. Each gain update requires an increase-key or decrease-key operations on the corresponding heap. Consequently, the complexity of rFM is \( O(2p \log n) = O(p \log n) \). This is equal to the complexity of the heap implementation of original FM where gains are stored in heaps instead of buckets.

Given these complexity values, the complexity of a single pass of rFM is \( O(n + n \log n + p \log n) = O(p \log n) \) since \( p \geq n \).

In our implementation, the space that rFM requires is modest. For each vertex, an additional gain value is stored compared to original FM. For each net, it is necessary to store an extra field that indicates the number of replicated vertices in addition to the number of non-replicated vertices that are in part \( A \) and \( B \).

### 4.2 rFM and Multilevel Framework

The multilevel framework for HP consists of 3 phases as mentioned in Chapter 2.3. The replication is achieved in the uncoarsening phase of the multilevel scheme where the refinement algorithm rFM is used as a replication tool. The coarsening and the initial partitioning phases are used as-is since they do not include the replication process.

At each level of the uncoarsening phase, we perform multiple passes to refine
the current bipartition. At the end of each level, the bipartition $\Pi^R_i$ on the coarser hypergraph $\mathcal{H}_i$ is projected back to the bipartition $\Pi^R_{i-1}$ on the finer hypergraph $\mathcal{H}_{i-1}$. The projection includes the decomposition of each super-vertex in $\mathcal{H}_i$ to its constituent vertices in $\mathcal{H}_{i-1}$. The decomposition of a non-replicated super-vertex in $\mathcal{H}_i$ results in multiple non-replicated vertices in $\mathcal{H}_{i-1}$. Similarly, the decomposition of a replicated super-vertex in $\mathcal{H}_i$ results in multiple replicated vertices in $\mathcal{H}_{i-1}$. The existence of replicated vertices does not disturb the projection process. Clearly, the decomposition of a replicated super-vertex to its constituent replicated vertices will not change the cut-state of the nets this replicated super-vertex is connected to. Furthermore, the single pin nets which are eliminated in the coarsening phase of the corresponding level will occur in the finer hypergraph. If this single pin is a replicated vertex, such nets will have only replicated vertices connected to it in the finer hypergraph and thus, the single pin nets will still be internal.

Unnecessary replicas tend to occur excessively at the beginning of each uncoarsening level due to the increase in the degrees of freedom after the projection of a coarser hypergraph to a finer hypergraph. Such replicas hamper the refinement and partitioning if they are not removed, since:

- They consume the given replication amount needlessly which may prevent the positive gain replications to be performed.
- In the construction of the new hypergraphs for further bipartitions, they can cause the new hypergraphs to become unnecessarily bigger.

In the operation selection, we give the unreplication of unnecessary replicas the highest priority (Section 4.1.2). By giving this operation the highest priority, the majority of the unnecessary replicas are eliminated at the beginning of each uncoarsening level.
4.3 Recursive Bipartitioning and Replica Selection

The $K$-way HP problem is generally solved with Recursive Bipartitioning framework (Chapter 2.3). In the RB scheme, firstly a 2-way partition of the initial hypergraph is obtained and then, the obtained parts are bipartitioned in a recursive manner until reaching $K$ parts. We use the same scheme to obtain multi-way partitioning of a given hypergraph for the replicated HP problem. After each bipartition, two new hypergraphs are constructed from the parts of the bipartition for further bipartitioning. In the construction of the new hypergraphs, the replicas of the replicated vertices become non-replicated vertices for the new hypergraphs and the necessary pins are placed for these vertices. Consequently, the weight and the number of pins of the resulting hypergraphs are greater when compared to the HP without replication.

4.3.1 Cut-net Splitting

In the RB framework, the cut-net splitting scheme is used in order to capture the connectivity cutsize metric. After each bipartitioning, two sub-hypergraphs, $\mathcal{H}_A$ and $\mathcal{H}_B$, are constructed from $\Pi^R = \{V_A, V_B\}$. The vertex sets of $\mathcal{H}_A$ and $\mathcal{H}_B$ are equivalent to $V_A$ and $V_B$ respectively. Clearly, the internal nets in $A$ will be in the net set of $\mathcal{H}_A$ and the internal nets in $B$ will be in the net set of $\mathcal{H}_B$. Each cut-net net $n_j \in \Pi^R$ is split into two nets, $n_j^A$ and $n_j^B$, where $\text{Pins}(n_j^A) = \text{Pins}(n_j) \cap V_A$ and $\text{Pins}(n_j^B) = \text{Pins}(n_j) \cap V_B$. Then, $n_j^A$ is added to the net list of $\mathcal{H}_A$ if $\sigma_A(n_j^A) > 1$ and $n_j^B$ is added to the net list of $\mathcal{H}_B$ if $\sigma_B(n_j^B) > 1$. Clearly, $\sigma_B(n_j^A) = 0$ and $\sigma_A(n_j^B) = 0$. Single pin nets are eliminated in splitting since they cannot contribute to cutsize in further bipartitions.

The cut-net splitting scheme is extended to include pins to the replicas of the replicated vertices. In the cut-net splitting scheme, where there are replicated vertices, we need to add pins to the replicas of the replicated vertices in order to preserve the flexibility of performing move or replication operations on them in
the newly constructed hypergraphs. After the split of a cut-net $n_j$, a pin is added for each replica of the replicated vertex connected by $n_j$ in part $A$ if $\sigma_A(n_j^A) > 1$ and for each replica of $n_j$ in part $B$ if $\sigma_B(n_j^B) > 1$. We do not add pin(s) to the replica(s) in part $A$ if $\sigma_A(n_j^A) < 2$ since $n_j^A$ cannot be cut in further bipartitions in $\mathcal{H}_A$. Similarly, we do not add pin(s) to the replica(s) in part $B$ if $\sigma_B(n_j^B) < 2$ since $n_j^B$ cannot be cut in further bipartitions in $\mathcal{H}_B$. Figs. 4.9 and 4.10 show the splitting two different cut-nets. In Fig. 4.9, the vertex $v_r$ is replicated and the pins for the replicas of $v_r$ need to be added for the split nets since $\sigma_A(n_j^A) = 3$ and $\sigma_B(n_j^B) = 2$. In Fig. 4.10, there are two replicated vertices, $v_r$ and $v_s$. After the splitting of $n_j$, we do not need to add the pins to the replicas of $v_r$ and $v_s$ in part $A$ since $\sigma_A(n_j^A) = 1$. However, we need to add pins to the replicas of $v_r$ and $v_s$ in part $B$ since $\sigma_B(n_j^B) = 2$. 

Figure 4.9: Cut-net splitting, no pins of net $n_j$ are discarded.

Figure 4.10: Cut-net splitting, the pins of net $n_j$ in part $A$ are discarded.
4.3.2 Replica Selection

The replication of a vertex \( v_i \) brings the problem of selecting replicas of \( v_i \) for each net it is connected by. If a net \( n_j \) connects replicated vertices, we need to decide which replicas of these replicated vertices will be used by \( n_j \). This is required for a couple of reasons: (i) the cut computation of the final partition and; (ii) the investigated real world problem may enforce the nets to make a choice from which parts their replicas will be used. We propose a simple replica selection technique.

The basic motivation behind this technique is not to increase the cutsize with careless replica selection. Fig. 4.11 shows various replica selection alternatives for \( n_j \) in a 3-way partition. There are two replicated vertices connected to \( n_j \), \( v_r \) and \( v_s \), each having three replicas and, there is a non-replicated vertex connected to \( n_j \) in part \( P_2 \), \( v_n \). Clearly, \( \lambda_j \) will be at least one because of the non-replicated vertex in \( P_2 \). If we select the replica of \( v_r \) in \( P_1 \) and the replica of \( v_s \) in \( P_3 \) for \( n_j \) as in Fig. 4.11a, \( n_j \) is cut \( (\lambda_j = 3) \) and the contribution of \( n_j \) to the cutsize is \( 2c(n_j) \). Another selection alternative may be to select the both replicas of \( v_r \) and \( v_s \) in \( P_3 \) as in Fig. 4.11b. In this case, \( n_j \) is cut \( (\lambda_j = 2) \) and the contribution of \( n_j \) to the cutsize is \( c(n_j) \). The logical selection alternative is seen in Fig. 4.11c, where the both replicas of \( v_r \) and \( v_s \) are selected from \( P_2 \), from the part of \( v_n \). In this replica selection alternative, \( n_j \) is uncut \( (\lambda_j = 1) \) and \( n_j \) does not contribute to the cutsize of the partition. This example shows how replica selection can be crucial in computing cutsize of given partitions.

Our replica selection technique consists of evaluating each net’s replicas and making a decision about which one to use after obtaining a K-way partitioning. The replica selection decision is based on the pin distributions of nets. Consider a net \( n_j \) and a replicated vertex \( v_i \) connected by it that has \( n \) replicas, \( r_1, \ldots, r_n \), we are to pick one of \( n \) replicas of \( v_i \) for \( n_j \). For each \( r_i \), we count the number of non-replicated and replicated vertices connected by \( n_j \) which are in the same part with \( r_i \). Then, we pick the replica with the highest number of non-replicated vertices. Selecting such a replica will not increase the cutsize since the part of the selected replica already contributes to cutsize for \( n_j \) because of the non-replicated vertices connected to \( n_j \) in that part. If the number of non-replicated vertices
(a) Careless replica selection for $n_j$ ($\lambda_j = 3$).
(b) Another careless replica selection for $n_j$ ($\lambda_j = 2$).
(c) Careful replica selection for $n_j$ ($\lambda_j = 1$).

Figure 4.11: Three replica selection alternatives for $n_j$. 
connected to $n_j$ which are in the same part with $r_i$ is zero for all replicas, we pick the replica with the highest number of replicated vertices which are connected to $n_j$ and in the same part with $r_i$. In this way, we assure $n_j$ to select its replicas from the part which possesses the highest number of replicated vertices connected to $n_j$. If this value is zero too, then we randomly pick one of the replicas of $v_i$ for $n_j$. Fig. 4.12 shows an example of this selection technique on nets $n_j$ and $n_k$ in a 4-way partition. There are two replicated vertices, $v_r$, $v_s$ and, three non-replicated vertices, $v_m$, $v_n$, $v_p$. After the replica selections are done for $n_j$ and $n_k$ with respect to the criteria mentioned above, $\lambda_j = 3$ and $\lambda_k = 2$.

![Figure 4.12: Replica selection for $n_j$ ($\lambda_j = 3$) and $n_k$ ($\lambda_k = 2$).](image)

### 4.3.3 Replication Amount Distribution

The RB scheme consists of multiple bipartitions. The replication amount used in each bipartitioning can have an effect on the cutsize of the final partition. We try 2 different replication amount distribution schemes in this work:

- **Level-wise Replication.** The replication amount is distributed evenly among the levels of the RB. Firstly, the total replication amount is divided by the number of levels, $\lg_2 K$. Then, for each specific level, the replication amount is evenly distributed among the hypergraphs in this level.

- **Bisection-wise Replication.** In this scheme, each bipartitioning possesses the same amount of replication.
Chapter 5

Experimental Results

In this chapter, we conduct experiments to test the performance of the proposed replication scheme on the cutsize and balance of the partitions. Firstly, we briefly explain the integration of our replication scheme into the multilevel hypergraph partitioning tool PaToH [15] and give details of the experimental setup. Then, we discuss the properties of the datasets used in the experiments. In the experiments, we compare two important quality metrics, cutsize and imbalance, of the partitions with and without replication. We also evaluate the gradient methodology described in the previous chapter in our experiments.

5.1 Experimental Setup

The proposed replication scheme is implemented and integrated into the multilevel HP tool PaToH. This version of PaToH is capable of vertex replication and, we call it replicated PaToH (repl-PaToH). As mentioned in previous chapters, replication is achieved in the uncoarsening phase of the multilevel methodology. In the coarsening and the initial partitioning phases, PaToH is used as-is, however, the uncoarsening phase is written from scratch. In our experiments, we use the same parameters for PaToH and repl-PaToH in the coarsening and the initial
partitioning phases. We use agglomerative clustering (absorption clustering using pins) and greedy hypergraph growing partition algorithms in the coarsening and initial partitioning phases, respectively. The initial partitioning algorithm is run multiple times and the bipartition with minimum cutsize is selected for the uncoarsening phase.

The parameters that are used in the uncoarsening phase are worth to mention for both versions:

- **Refinement algorithm**, is the algorithm that is used as the refinement algorithm in the uncoarsening phase. For PaToH, boundary FM (BFM) and boundary KL (BKL) algorithm is used. For repl-PaToH, we use the rFM algorithm described in the previous chapter.

- **Initial and final imbalance**, are the values that must be assured at the beginning and at the end of each uncoarsening phase, respectively. Maintaining a loose initial imbalance value may help FM based heuristics to work better since at the beginning levels, the average weight of a single vertex is greater and performing an operation on a vertex may be prevented due to tight size constraints on the parts. Thus, the initial imbalance is set to 0.12 and the final imbalance is set to 0.10 for PaToH and repl-PaToH.

- **Number of passes**, is the value that how many times the refinement algorithm is run at each level of the uncoarsening phase. We set this parameter to 3 for PaToH and repl-PaToH in our experiments.

- **Window size**, is the number of operations that is allowed to be performed which do not improve the cutsize. The window size is set to 100 for both PaToH and repl-PaToH.

The experiments are conducted on a 4 × AMD Six-Core Opteron, each core having a clock frequency of 2.1 GHz, 64 KB L1 Instruction and Data cache, 512 KB L2 cache and 6 MB shared L3 cache. Each processor has a 32 GB of memory which makes a total of 128 GB memory. The implementation is done
Table 5.1: The properties of the data sets used in our experiments.

<table>
<thead>
<tr>
<th>Data set</th>
<th># vertices</th>
<th># nets</th>
<th># pins</th>
<th>Avg. net deg.</th>
<th>Avg. net weight</th>
<th>Avg. vertex weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>10141</td>
<td>17369</td>
<td>52965</td>
<td>3.05</td>
<td>13.36</td>
<td>48.76</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>34222</td>
<td>46056</td>
<td>149493</td>
<td>3.25</td>
<td>26.86</td>
<td>47.11</td>
</tr>
<tr>
<td>New Mexico</td>
<td>448959</td>
<td>510477</td>
<td>1682429</td>
<td>3.30</td>
<td>17.32</td>
<td>43.64</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>4465</td>
<td>7886</td>
<td>22085</td>
<td>2.80</td>
<td>15.06</td>
<td>42.62</td>
</tr>
<tr>
<td>Oregon</td>
<td>507212</td>
<td>574353</td>
<td>1844579</td>
<td>3.21</td>
<td>17.96</td>
<td>41.96</td>
</tr>
<tr>
<td>San Francisco</td>
<td>166558</td>
<td>197630</td>
<td>654990</td>
<td>3.31</td>
<td>18.34</td>
<td>44.99</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>17444</td>
<td>26225</td>
<td>80785</td>
<td>3.08</td>
<td>16.66</td>
<td>46.17</td>
</tr>
<tr>
<td>Washington</td>
<td>548901</td>
<td>613598</td>
<td>1980535</td>
<td>3.23</td>
<td>21.77</td>
<td>42.01</td>
</tr>
<tr>
<td>Wyoming</td>
<td>254648</td>
<td>302494</td>
<td>984030</td>
<td>3.25</td>
<td>17.53</td>
<td>43.85</td>
</tr>
<tr>
<td>CalGovernerRecall</td>
<td>172556</td>
<td>30805</td>
<td>3545766</td>
<td>115.10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Facebook</td>
<td>4618974</td>
<td>66568</td>
<td>14277456</td>
<td>214.48</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Stanford</td>
<td>281903</td>
<td>281903</td>
<td>2312497</td>
<td>8.20</td>
<td>1.00</td>
<td>8.20</td>
</tr>
</tbody>
</table>

in C programming language and all files are compiled with gcc –O3 optimization flag enabled.

5.2 Datasets

In the experiments, we test our algorithm on various datasets from spatial network and information retrieval fields. Various characteristics of these datasets are depicted in Table 5.1 such as number of vertices, nets and pins, average net degree, average net and vertex weight. There are 9 data sets for spatial networks that include data sets from US Department of Transportation [23] (California HPN dataset), US Tiger/Line [16] (Minnesota7 that includes data from 7 countries, New Mexico, Oregon, San Francisco, Washington and Wyoming datasets) and Brinkhoff’s network data generator [9] (Oldenburg and San Joaquin datasets). There are 3 data sets for information retrieval. Two of them are crawled using Stanford WebBase Project [17, 54] (Facebook and CalGovernerRecall datasets) and the other one is from University of Florida Sparse Matrix Collection [19] (Stanford dataset which depicts the Stanford web graph).

The basic difference between the spatial network and IR datasets is clearly
the average net degree where in IR datasets this value is much higher. In the hypergraph models for road networks, the junctions are considered as both nets and vertices and each junction has an average of about three roads connected to it when we consider the datasets in Table 5.1. However, in IR based hypergraph models, generally the nets represent the documents and the pins of a net represent the terms/links in that document. From this perspective, it is obvious that average degree of a net for the hypergraph models in IR will be higher than those in spatial networks.

5.3 Results

The results include two important quality metrics in HP: cutsize and imbalance. We illustrate the results for $K = 32, 64, 128, 256$ and $\rho = 0.05, 0.10, 0.20$. Firstly, we compare PaToH and two replication amount distribution schemes (bisection-wise and level-wise) mentioned in Chapter 4. Then, rFM heuristic and the gradient methodology is compared. Each instance is run 10 times and the average of these runs is shown in the results.

Tables 5.2, 5.3, 5.4 and 5.5 show the cutsize and imbalance values of PaToH and repl-PaToH with two different replication schemes, bisection-wise and level-wise replication. Figs. 5.1, 5.2, 5.3 and 5.4 illustrate the improvement in the cutsize of two different replication schemes which are represented in the mentioned tables. In spatial network datasets, the improvement in the cutsize is greater than the improvement in the IR datasets. This is mainly due to the difference of average net degrees between these datasets. In IR datasets, the average net degree is high and saving a net from cut is harder compared to the nets in spatial network datasets. For example, in Table 5.3, for spatial network datasets, the average improvement of repl-PaToH (bisection-wise) for $\rho = 0.05, 0.10$ and, 0.20 is 56.52, 62.55 and, 62.32, respectively whereas for IR datasets, these values are 13.22, 17.87 and, 22.62, respectively. The values in other tables indicate similar results where the improvement in the cutsize in IR datasets is lower. As $\rho$ grows higher, generally, the cutsize decreases for a specific value of $K$ since
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>13810</td>
<td>5.72</td>
<td>6501</td>
<td>52.92</td>
<td>4.78</td>
<td>8791</td>
<td>36.34</td>
<td>5.18</td>
<td>0.10</td>
<td>13896</td>
<td>5.49</td>
<td>3599</td>
<td>74.10</td>
<td>5.29</td>
<td>6559</td>
<td>52.79</td>
<td>5.59</td>
<td>0.20</td>
<td>13921</td>
<td>6.21</td>
<td>3368</td>
<td>75.80</td>
<td>7.26</td>
<td>4693</td>
<td>66.28</td>
<td>5.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>20542</td>
<td>5.11</td>
<td>4252</td>
<td>79.30</td>
<td>5.98</td>
<td>6868</td>
<td>66.56</td>
<td>4.55</td>
<td>0.10</td>
<td>20279</td>
<td>5.14</td>
<td>4969</td>
<td>75.49</td>
<td>4.04</td>
<td>6335</td>
<td>68.76</td>
<td>4.09</td>
<td>0.20</td>
<td>20118</td>
<td>5.28</td>
<td>7000</td>
<td>65.20</td>
<td>5.08</td>
<td>7081</td>
<td>64.80</td>
<td>3.79</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>23241</td>
<td>6.72</td>
<td>8900</td>
<td>61.70</td>
<td>4.54</td>
<td>7854</td>
<td>69.08</td>
<td>4.02</td>
<td>0.10</td>
<td>23210</td>
<td>6.16</td>
<td>10033</td>
<td>56.77</td>
<td>2.64</td>
<td>8225</td>
<td>67.60</td>
<td>2.70</td>
<td>0.20</td>
<td>23163</td>
<td>6.18</td>
<td>11949</td>
<td>48.41</td>
<td>4.98</td>
<td>3658</td>
<td>64.84</td>
<td>5.10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>10348</td>
<td>6.34</td>
<td>5751</td>
<td>44.42</td>
<td>5.45</td>
<td>6701</td>
<td>35.24</td>
<td>5.37</td>
<td>0.10</td>
<td>10325</td>
<td>5.98</td>
<td>3247</td>
<td>68.55</td>
<td>5.28</td>
<td>5654</td>
<td>51.05</td>
<td>4.91</td>
<td>0.20</td>
<td>10406</td>
<td>6.05</td>
<td>2665</td>
<td>74.38</td>
<td>4.98</td>
<td>3658</td>
<td>64.84</td>
<td>5.10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>29125</td>
<td>5.11</td>
<td>11584</td>
<td>60.22</td>
<td>4.67</td>
<td>12790</td>
<td>56.08</td>
<td>4.02</td>
<td>0.10</td>
<td>28550</td>
<td>5.14</td>
<td>13071</td>
<td>54.21</td>
<td>2.64</td>
<td>14305</td>
<td>49.89</td>
<td>1.86</td>
<td>0.20</td>
<td>28704</td>
<td>6.18</td>
<td>13958</td>
<td>48.41</td>
<td>1.64</td>
<td>17286</td>
<td>39.77</td>
<td>1.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>25402</td>
<td>6.40</td>
<td>6827</td>
<td>73.12</td>
<td>4.13</td>
<td>7854</td>
<td>69.08</td>
<td>4.02</td>
<td>0.10</td>
<td>25393</td>
<td>6.48</td>
<td>7938</td>
<td>68.73</td>
<td>2.92</td>
<td>8225</td>
<td>67.60</td>
<td>2.70</td>
<td>0.20</td>
<td>25361</td>
<td>7.01</td>
<td>9124</td>
<td>64.02</td>
<td>2.64</td>
<td>10035</td>
<td>60.43</td>
<td>1.81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>17723</td>
<td>6.20</td>
<td>5898</td>
<td>66.72</td>
<td>6.02</td>
<td>8617</td>
<td>51.37</td>
<td>5.09</td>
<td>0.10</td>
<td>17417</td>
<td>5.57</td>
<td>4141</td>
<td>76.22</td>
<td>6.29</td>
<td>6324</td>
<td>63.69</td>
<td>4.72</td>
<td>0.20</td>
<td>17582</td>
<td>6.55</td>
<td>4631</td>
<td>73.66</td>
<td>5.54</td>
<td>5328</td>
<td>69.69</td>
<td>4.96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>27088</td>
<td>6.48</td>
<td>8611</td>
<td>68.21</td>
<td>4.06</td>
<td>9448</td>
<td>65.12</td>
<td>3.05</td>
<td>0.10</td>
<td>27114</td>
<td>5.99</td>
<td>10266</td>
<td>62.35</td>
<td>3.26</td>
<td>11859</td>
<td>56.26</td>
<td>1.63</td>
<td>0.20</td>
<td>27251</td>
<td>6.33</td>
<td>12186</td>
<td>55.28</td>
<td>2.00</td>
<td>13076</td>
<td>52.01</td>
<td>1.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>28823</td>
<td>6.08</td>
<td>11998</td>
<td>58.37</td>
<td>3.73</td>
<td>14121</td>
<td>51.00</td>
<td>3.50</td>
<td>0.10</td>
<td>28746</td>
<td>6.43</td>
<td>14461</td>
<td>49.69</td>
<td>2.70</td>
<td>15399</td>
<td>46.43</td>
<td>1.85</td>
<td>0.20</td>
<td>28747</td>
<td>6.42</td>
<td>15907</td>
<td>44.66</td>
<td>1.51</td>
<td>17784</td>
<td>38.10</td>
<td>0.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>6.15</td>
<td>62.78</td>
<td>4.82</td>
<td>53.99</td>
<td>4.18</td>
<td>56.18</td>
<td>3.24</td>
<td>0.10</td>
<td>5.92</td>
<td>65.12</td>
<td>3.92</td>
<td>56.18</td>
<td>3.24</td>
<td>0.20</td>
<td>6.33</td>
<td>61.42</td>
<td>3.63</td>
<td>55.23</td>
<td>2.94</td>
<td>56.18</td>
<td>3.24</td>
<td>0.20</td>
<td>6.33</td>
<td>61.42</td>
<td>3.63</td>
<td>55.23</td>
<td>2.94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CalGovernorRecall</td>
<td>0.05</td>
<td>90677</td>
<td>3.19</td>
<td>87866</td>
<td>3.10</td>
<td>9.10</td>
<td>90147</td>
<td>0.58</td>
<td>8.18</td>
<td>0.10</td>
<td>92010</td>
<td>3.66</td>
<td>84604</td>
<td>8.04</td>
<td>9.89</td>
<td>87368</td>
<td>5.04</td>
<td>8.47</td>
<td>0.20</td>
<td>90956</td>
<td>3.65</td>
<td>81315</td>
<td>9.70</td>
<td>9.56</td>
<td>85353</td>
<td>5.22</td>
<td>9.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>173581</td>
<td>0.00</td>
<td>159872</td>
<td>7.89</td>
<td>9.06</td>
<td>168977</td>
<td>2.65</td>
<td>8.99</td>
<td>0.10</td>
<td>174658</td>
<td>0.00</td>
<td>158727</td>
<td>9.12</td>
<td>8.59</td>
<td>160176</td>
<td>8.29</td>
<td>8.57</td>
<td>0.20</td>
<td>176880</td>
<td>0.00</td>
<td>153042</td>
<td>13.47</td>
<td>7.51</td>
<td>154873</td>
<td>12.44</td>
<td>8.29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>5552</td>
<td>4.62</td>
<td>3779</td>
<td>31.93</td>
<td>9.36</td>
<td>4138</td>
<td>25.46</td>
<td>8.95</td>
<td>0.10</td>
<td>5563</td>
<td>4.71</td>
<td>3302</td>
<td>40.64</td>
<td>8.82</td>
<td>3847</td>
<td>30.84</td>
<td>7.98</td>
<td>0.20</td>
<td>5454</td>
<td>4.94</td>
<td>3012</td>
<td>44.74</td>
<td>9.29</td>
<td>3665</td>
<td>32.76</td>
<td>8.37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>2.60</td>
<td>14.31</td>
<td>9.17</td>
<td>9.56</td>
<td>8.71</td>
<td>14.72</td>
<td>8.94</td>
<td>0.10</td>
<td>2.79</td>
<td>19.27</td>
<td>9.43</td>
<td>14.72</td>
<td>8.94</td>
<td>0.20</td>
<td>2.86</td>
<td>22.64</td>
<td>8.79</td>
<td>16.81</td>
<td>8.57</td>
<td>14.72</td>
<td>8.94</td>
<td>0.20</td>
<td>2.86</td>
<td>22.64</td>
<td>8.79</td>
<td>16.81</td>
<td>8.57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVERAGE (All)</td>
<td>0.05</td>
<td>5.26</td>
<td>50.66</td>
<td>5.91</td>
<td>42.88</td>
<td>5.32</td>
<td>45.82</td>
<td>4.66</td>
<td>0.10</td>
<td>5.14</td>
<td>53.66</td>
<td>5.30</td>
<td>45.82</td>
<td>4.66</td>
<td>0.20</td>
<td>5.47</td>
<td>51.72</td>
<td>4.92</td>
<td>45.82</td>
<td>4.66</td>
<td>45.82</td>
<td>4.66</td>
<td>0.20</td>
<td>5.47</td>
<td>51.72</td>
<td>4.92</td>
<td>45.82</td>
<td>4.66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: The cut and imbalance values for $K = 32$. 
<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\rho$</th>
<th>PaToH</th>
<th>repl-PaToH, bisection-wise</th>
<th>repl-PaToH, level-wise</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>24882</td>
<td>5.07</td>
<td>16074</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>24847</td>
<td>5.04</td>
<td>10208</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>25121</td>
<td>5.57</td>
<td>6769</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>38226</td>
<td>5.09</td>
<td>10623</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>38176</td>
<td>5.60</td>
<td>7851</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>37664</td>
<td>6.34</td>
<td>9889</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>40443</td>
<td>6.57</td>
<td>15469</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>40422</td>
<td>6.20</td>
<td>17420</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>40443</td>
<td>6.50</td>
<td>20233</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>16866</td>
<td>5.18</td>
<td>11929</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>16767</td>
<td>5.86</td>
<td>7546</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>16690</td>
<td>6.02</td>
<td>5016</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>47694</td>
<td>6.52</td>
<td>18868</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>47565</td>
<td>6.94</td>
<td>21179</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>47592</td>
<td>6.54</td>
<td>23926</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>44665</td>
<td>6.39</td>
<td>13856</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>44815</td>
<td>6.58</td>
<td>13755</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>44603</td>
<td>6.33</td>
<td>15072</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>29823</td>
<td>6.00</td>
<td>14955</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>29838</td>
<td>5.74</td>
<td>8278</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>29939</td>
<td>5.61</td>
<td>7643</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>47565</td>
<td>6.09</td>
<td>16177</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>47027</td>
<td>5.76</td>
<td>17646</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>47639</td>
<td>5.95</td>
<td>20502</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>47327</td>
<td>6.12</td>
<td>19270</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>47459</td>
<td>6.33</td>
<td>22196</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>47671</td>
<td>6.57</td>
<td>24706</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>5.89</td>
<td>56.52</td>
<td>5.05</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>6.01</td>
<td>62.55</td>
<td>4.67</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>6.06</td>
<td>62.32</td>
<td>4.84</td>
</tr>
<tr>
<td>CalGovernerRecall</td>
<td>0.05</td>
<td>139097</td>
<td>4.21</td>
<td>135886</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>138494</td>
<td>4.55</td>
<td>132041</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>138909</td>
<td>4.71</td>
<td>127518</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>225148</td>
<td>0.73</td>
<td>205894</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>226226</td>
<td>0.47</td>
<td>197657</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>225076</td>
<td>0.66</td>
<td>190866</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>9546</td>
<td>4.33</td>
<td>6794</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>9485</td>
<td>4.39</td>
<td>6038</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>9475</td>
<td>4.51</td>
<td>5248</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>3.09</td>
<td>13.22</td>
<td>9.25</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>3.13</td>
<td>17.87</td>
<td>9.39</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>3.29</td>
<td>22.62</td>
<td>9.50</td>
</tr>
<tr>
<td>AVERAGE (All)</td>
<td>0.05</td>
<td>5.19</td>
<td>45.70</td>
<td>6.10</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>5.29</td>
<td>51.38</td>
<td>5.85</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>5.37</td>
<td>52.39</td>
<td>6.01</td>
</tr>
</tbody>
</table>

Table 5.3: The cut and imbalance values for $K = 64$. 
<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\rho$</th>
<th>PaToH Cut</th>
<th>PaToH Imb.</th>
<th>repl-PaToH bisection-wise Cut</th>
<th>repl-PaToH bisection-wise Imb.</th>
<th>repl-PaToH level-wise Cut</th>
<th>repl-PaToH level-wise Imb.</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>41424</td>
<td>5.95</td>
<td>23458</td>
<td>43.21</td>
<td>30324</td>
<td>26.58</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>41307</td>
<td>5.86</td>
<td>14322</td>
<td>65.45</td>
<td>23888</td>
<td>42.58</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>41464</td>
<td>5.62</td>
<td>13422</td>
<td>65.45</td>
<td>23888</td>
<td>42.58</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>67591</td>
<td>5.93</td>
<td>28680</td>
<td>57.56</td>
<td>42993</td>
<td>36.39</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>67641</td>
<td>5.61</td>
<td>18017</td>
<td>73.36</td>
<td>33997</td>
<td>51.06</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>67517</td>
<td>5.77</td>
<td>14903</td>
<td>77.92</td>
<td>25824</td>
<td>61.75</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>68009</td>
<td>5.95</td>
<td>24827</td>
<td>63.19</td>
<td>27926</td>
<td>58.93</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>67586</td>
<td>6.25</td>
<td>27821</td>
<td>58.83</td>
<td>30373</td>
<td>55.06</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>67806</td>
<td>6.53</td>
<td>31490</td>
<td>53.59</td>
<td>35969</td>
<td>46.99</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>76044</td>
<td>5.62</td>
<td>23458</td>
<td>17.37</td>
<td>23814</td>
<td>11.94</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>76016</td>
<td>5.97</td>
<td>18224</td>
<td>32.54</td>
<td>20995</td>
<td>22.28</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>76069</td>
<td>6.13</td>
<td>11130</td>
<td>58.63</td>
<td>16478</td>
<td>38.76</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>76467</td>
<td>5.62</td>
<td>28145</td>
<td>63.19</td>
<td>33560</td>
<td>56.11</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>76594</td>
<td>5.84</td>
<td>32118</td>
<td>58.06</td>
<td>35731</td>
<td>53.35</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>76557</td>
<td>6.37</td>
<td>35735</td>
<td>53.01</td>
<td>39863</td>
<td>47.58</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>76922</td>
<td>6.43</td>
<td>20551</td>
<td>72.96</td>
<td>35533</td>
<td>53.25</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>76634</td>
<td>6.71</td>
<td>23697</td>
<td>70.81</td>
<td>39126</td>
<td>61.73</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>76264</td>
<td>6.22</td>
<td>20000</td>
<td>65.90</td>
<td>28869</td>
<td>62.14</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>49140</td>
<td>5.46</td>
<td>31822</td>
<td>35.24</td>
<td>37108</td>
<td>24.48</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>49292</td>
<td>5.59</td>
<td>20612</td>
<td>58.18</td>
<td>29915</td>
<td>39.31</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>49106</td>
<td>5.71</td>
<td>13678</td>
<td>72.14</td>
<td>22753</td>
<td>53.66</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>84776</td>
<td>5.49</td>
<td>26875</td>
<td>68.29</td>
<td>33183</td>
<td>62.44</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>85026</td>
<td>6.27</td>
<td>29485</td>
<td>65.32</td>
<td>33335</td>
<td>60.79</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>84980</td>
<td>6.28</td>
<td>33831</td>
<td>60.18</td>
<td>37748</td>
<td>55.58</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>75084</td>
<td>5.68</td>
<td>27924</td>
<td>62.80</td>
<td>35813</td>
<td>52.30</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>75483</td>
<td>5.92</td>
<td>31904</td>
<td>57.73</td>
<td>37585</td>
<td>50.92</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>75303</td>
<td>5.57</td>
<td>36891</td>
<td>51.00</td>
<td>42084</td>
<td>44.11</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>49.7</td>
<td>5.87</td>
<td>31.24</td>
<td>5.40</td>
<td>41.24</td>
<td>4.77</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>6.11</td>
<td>5.76</td>
<td>7.29</td>
<td>5.42</td>
<td>46.71</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>6.05</td>
<td>5.98</td>
<td>9.34</td>
<td>5.22</td>
<td>50.35</td>
<td>4.06</td>
</tr>
<tr>
<td>CalGovernorRecall</td>
<td>0.05</td>
<td>206593</td>
<td>5.00</td>
<td>202054</td>
<td>3.13</td>
<td>209112</td>
<td>-0.24</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>210678</td>
<td>4.89</td>
<td>199267</td>
<td>5.41</td>
<td>204878</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>208122</td>
<td>5.04</td>
<td>196444</td>
<td>4.55</td>
<td>208593</td>
<td>0.78</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>290153</td>
<td>1.69</td>
<td>259257</td>
<td>10.64</td>
<td>262754</td>
<td>9.44</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>289703</td>
<td>1.30</td>
<td>256722</td>
<td>10.54</td>
<td>261865</td>
<td>8.74</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>285886</td>
<td>1.37</td>
<td>251918</td>
<td>11.88</td>
<td>264446</td>
<td>5.75</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>14739</td>
<td>4.56</td>
<td>11479</td>
<td>22.11</td>
<td>12517</td>
<td>15.07</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>14813</td>
<td>4.72</td>
<td>10354</td>
<td>30.10</td>
<td>11708</td>
<td>20.96</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>14804</td>
<td>4.49</td>
<td>9202</td>
<td>37.84</td>
<td>10848</td>
<td>26.92</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>3.75</td>
<td>11.96</td>
<td>9.23</td>
<td>8.09</td>
<td>8.64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>3.64</td>
<td>15.35</td>
<td>9.46</td>
<td>10.80</td>
<td>8.59</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>3.64</td>
<td>18.09</td>
<td>9.65</td>
<td>11.15</td>
<td>9.19</td>
<td></td>
</tr>
<tr>
<td>AVERAGE (All)</td>
<td>0.05</td>
<td>5.34</td>
<td>41.78</td>
<td>6.36</td>
<td>32.95</td>
<td>5.74</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>5.49</td>
<td>47.01</td>
<td>6.43</td>
<td>37.73</td>
<td>5.16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>5.45</td>
<td>51.01</td>
<td>6.65</td>
<td>40.55</td>
<td>5.34</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: The cut and imbalance values for $K = 128$. 
### Table 5.5: The cut and imbalance values for $K = 256$.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\rho$</th>
<th>PaToH</th>
<th>repl-PaToH, bisection-wise</th>
<th>repl-PaToH, level-wise</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>66204</td>
<td>57949 12.46 5.93</td>
<td>60868 8.05 5.54</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>66342</td>
<td>47508 28.38 6.79</td>
<td>55299 16.64 5.26</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>66326</td>
<td>35506 49.48 7.53</td>
<td>46924 30.60 6.09</td>
</tr>
<tr>
<td>Minnesota</td>
<td>0.05</td>
<td>115295</td>
<td>5863 58.89 6.07</td>
<td>52043 54.86 4.66</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>115618</td>
<td>43859 5.19 4.40</td>
<td>48228 50.83 3.41</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>115198</td>
<td>29059 30.60 6.26</td>
<td>55411 51.89 6.70</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>115295</td>
<td>5863 58.89 6.07</td>
<td>52043 54.86 4.66</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>115618</td>
<td>43859 5.19 4.40</td>
<td>48228 50.83 3.41</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>115198</td>
<td>29059 30.60 6.26</td>
<td>55411 51.89 6.70</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>44058</td>
<td>6280 3.12 7.37</td>
<td>41283 6.29 5.53</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>44113</td>
<td>39742 6.58 6.11</td>
<td>38277 13.22 5.51</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>44063</td>
<td>35483 19.47 7.31</td>
<td>33825 23.23 6.25</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>123780</td>
<td>40319 67.41 6.01</td>
<td>55504 55.15 4.36</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>123141</td>
<td>47350 61.54 5.14</td>
<td>55977 54.54 5.39</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>123376</td>
<td>52547 34.84 4.84</td>
<td>59520 51.75 4.19</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>125656</td>
<td>42415 66.22 5.95</td>
<td>71058 43.45 4.63</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>125447</td>
<td>41045 72.86 7.13</td>
<td>55649 55.63 4.59</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>125553</td>
<td>38064 69.25 7.17</td>
<td>47008 62.55 5.68</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>79818</td>
<td>61244 23.27 6.24</td>
<td>68954 13.61 5.46</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>79766</td>
<td>46218 42.03 6.21</td>
<td>58662 26.45 4.84</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>80077</td>
<td>28480 64.43 8.58</td>
<td>46752 41.61 5.46</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>144799</td>
<td>40398 72.16 5.32</td>
<td>58844 59.36 4.17</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>144929</td>
<td>47888 67.19 5.61</td>
<td>54376 62.48 3.50</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>145185</td>
<td>52547 57.40 4.84</td>
<td>59881 58.75 3.77</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>120624</td>
<td>40955 66.04 5.62</td>
<td>66108 45.20 4.41</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>120407</td>
<td>45552 62.16 5.44</td>
<td>57317 52.39 3.69</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>120791</td>
<td>52501 56.53 4.93</td>
<td>60785 49.67 3.70</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>79818</td>
<td>61244 23.27 6.24</td>
<td>68954 13.61 5.46</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>79766</td>
<td>46218 42.03 6.21</td>
<td>58662 26.45 4.84</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>80077</td>
<td>28480 64.43 8.58</td>
<td>46752 41.61 5.46</td>
</tr>
<tr>
<td>CalGovernorRecall</td>
<td>0.05</td>
<td>305429</td>
<td>301908 1.15 9.87</td>
<td>300517 1.33 8.89</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>304936</td>
<td>300748 1.37 9.93</td>
<td>315186 -3.36 9.77</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>304065</td>
<td>298685 1.76 10.67 1.56</td>
<td>308817 10.06</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>367075</td>
<td>343711 6.36 10.80 6.00</td>
<td>345023 9.59 5.59</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>370908</td>
<td>337817 8.92 10.15 5.87</td>
<td>349106 9.63 5.99</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>372214</td>
<td>340130 8.61 10.81 9.22</td>
<td>337885 9.93 9.93</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>22233</td>
<td>18883 15.06 8.15</td>
<td>20072 7.11 7.87</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>22121</td>
<td>17022 23.05 10.42 5.67</td>
<td>18831 14.87 9.01</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>22200</td>
<td>15462 30.35 10.34 9.51</td>
<td>17867 10.42 10.42</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>0.05</td>
<td>5.83</td>
<td>46.66 6.06</td>
<td>34.52 8.78</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>5.85</td>
<td>52.24 6.11</td>
<td>41.94 9.47</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>5.99</td>
<td>57.80 6.60</td>
<td>46.97 10.14</td>
</tr>
<tr>
<td>AVERAGE (All)</td>
<td>0.05</td>
<td>5.50</td>
<td>36.88 6.94</td>
<td>27.09 5.86</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>5.46</td>
<td>41.96 7.13</td>
<td>32.91 5.71</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>5.57</td>
<td>46.15 7.60</td>
<td>37.49 6.27</td>
</tr>
</tbody>
</table>
with more replication, more nets can be saved from the cut. However, in some datasets, the opposite of this can happen where as \( \rho \) increases, the cutsize can decrease. For example in Table 5.4 for Oregon dataset with level-wise replication, the improvement in the cutsize for \( \rho = 0.05, 0.10 \) and, 0.20 is 56.11, 53.35 and, 47.58, respectively. The main reason behind this anomaly is that if more replication amount is given to a hypergraph than the amount it needs, extra replication can lead to partitions with worse cutsize. In other words, \( \rho = 0.10 \) or 0.20 for the Oregon dataset gives more replication amount than this dataset needs and thus, the partitions have slightly worse cutsize values. For a specific \( \rho \), as the value of \( K \) increases, the improvement in the cutsize decreases. For instance, for \( \rho = 0.10 \), the average cutsize values of all datasets for \( K = 32, 64, 128 \) and, 256 are 53.66, 51.38, 47.01 and, 41.96, respectively. This is because as the number of parts increases, the possibility of a cut-net connecting more parts also increases. The imbalance values for repl-PaToH is close to PaToH for spatial network datasets while repl-PaToH has clearly higher imbalance values for IR datasets since in most of the cases, the given replication amount is not used uniformly in each bipartitioning for these datasets.

![Figure 5.1: Improvement at \( K = 32 \).](image)

Almost in all datasets, bisection-wise replication comes up with better cutsize values than the level-wise replication. In bisection-wise replication, the initial bipartitions have lower replication amount with respect to their total vertex weight.
CHAPTER 5. EXPERIMENTAL RESULTS

Figure 5.2: Improvement at $K = 64$.

Figure 5.3: Improvement at $K = 128$. 
whereas in level-wise replication, all bipartitions have equal replication amount. From this, we can conclude that performing more replication at deeper levels of the recursion tree can be more helpful in obtaining partitions with better cutsize. On the contrary, level-wise replication has generally better imbalance values. The reason behind this is that each bipartition gets equal replication amount with respect to their total vertex weight and after performing replication, the total vertex weights of the hypergraphs with replicated vertices are close to each other.

Tables 5.6, 5.7, 5.8 and 5.9 show the cutsize values of rFM and rFM with gradient methodology (gradient-rFM). We used the same datasets and tested them for both bisection-wise and level-wise replication. In the tables, we compare the replication schemes separately for rFM and gradient-rFM. The lower cutsize value for a specific replication scheme is illustrated as bold. For instance, in Table 5.7 for the Washington dataset with $\rho = 0.05$, gradient-rFM (15803) has a lower cutsize than rFM (16177) for bisection-wise replication whereas rFM (17683) has lower cutsize value than gradient-rFM (18438) for level-wise replication. Generally, rFM performs better than gradient-rFM in spatial network datasets. However, in IR datasets, gradient-rFM performs better; especially in CalGovernerRecall dataset where for almost all $K$ values, gradient-rFM is superior to rFM. This can indicate that gradient-rFM may be well-suited to the hypergraphs with relatively higher average net degree.
### Table 5.6: The cut values for rFM and gradient rFM for $K = 32$.  

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\rho$</th>
<th>repl-PaToH, rFM</th>
<th>repl-PaToH, rFM-gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>6501</td>
<td>8791</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>3599</td>
<td>6559</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>3368</td>
<td>4693</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>4252</td>
<td>6868</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>4969</td>
<td>6335</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>7000</td>
<td>7081</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>8900</td>
<td>10434</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>10033</td>
<td>11800</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>11949</td>
<td>13633</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>5751</td>
<td>6701</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>3247</td>
<td>5054</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>2665</td>
<td>3658</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>11584</td>
<td>12790</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>13071</td>
<td>14305</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>13958</td>
<td>17286</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>6827</td>
<td>7854</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>7938</td>
<td>8225</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>9124</td>
<td>10035</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>5898</td>
<td>8617</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>4141</td>
<td>6324</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>4631</td>
<td>5328</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>8611</td>
<td>9448</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>10206</td>
<td>11859</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>12186</td>
<td>13076</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>11998</td>
<td>14121</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>14481</td>
<td>15399</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>15907</td>
<td>17794</td>
</tr>
<tr>
<td>CalGovernerRecall</td>
<td>0.05</td>
<td>87866</td>
<td>90147</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>84604</td>
<td>87368</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>81315</td>
<td>85353</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>159872</td>
<td>160176</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>158727</td>
<td>160176</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>153042</td>
<td>154873</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>3779</td>
<td>4138</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>3302</td>
<td>3847</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>3012</td>
<td>3665</td>
</tr>
<tr>
<td>Dataset</td>
<td>( \rho )</td>
<td>repl-PaToH, rFM</td>
<td>repl-PaToH, rFM-gradient</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>-----------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>16074</td>
<td>18888</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>10208</td>
<td>15319</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>6769</td>
<td>11181</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>10623</td>
<td>17793</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>7851</td>
<td>13493</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>9889</td>
<td>12669</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>15469</td>
<td>17505</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>17420</td>
<td>19423</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>20293</td>
<td>22222</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>11929</td>
<td>13273</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>7546</td>
<td>10423</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>5016</td>
<td>7557</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>18868</td>
<td>20735</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>21179</td>
<td>22759</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>23926</td>
<td>27158</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>11385</td>
<td>16438</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>13755</td>
<td>15903</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>15672</td>
<td>18197</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>14955</td>
<td>19189</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>8278</td>
<td>14543</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>7643</td>
<td>10830</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>16177</td>
<td>17683</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>17646</td>
<td>19986</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>20502</td>
<td>23064</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>19270</td>
<td>21753</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>22196</td>
<td>24557</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>24706</td>
<td>28956</td>
</tr>
<tr>
<td>CalGovernorRecall</td>
<td>0.05</td>
<td>135886</td>
<td>136342</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>132041</td>
<td>135345</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>127518</td>
<td>132349</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>205894</td>
<td>216265</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>197657</td>
<td>216342</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>190866</td>
<td>197758</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>6794</td>
<td>7503</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>6038</td>
<td>6777</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>5248</td>
<td>6459</td>
</tr>
</tbody>
</table>

Table 5.7: The cut values for rFM and gradient rFM for \( K = 64 \).
### Table 5.8: The cut values for rFM and gradient rFM for $K = 128.$

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\rho$</th>
<th>repl-PaToH, rFM</th>
<th>repl-PaToH, rFM-gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>31248</td>
<td>35072</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>23458</td>
<td>30324</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>14322</td>
<td>23808</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>28680</td>
<td>42993</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>18017</td>
<td>33997</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>14903</td>
<td>25824</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>24827</td>
<td>27926</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>27821</td>
<td>30373</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>31490</td>
<td>35969</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>22345</td>
<td>23814</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>18224</td>
<td>20995</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>11130</td>
<td>16478</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>28145</td>
<td>33560</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>32118</td>
<td>35731</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>35735</td>
<td>39863</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>20551</td>
<td>35533</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>22367</td>
<td>29326</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>26000</td>
<td>28869</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>31822</td>
<td>37198</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>20612</td>
<td>29915</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>13678</td>
<td>22753</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>26875</td>
<td>31838</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>29485</td>
<td>33335</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>33831</td>
<td>37748</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>27924</td>
<td>35813</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>31904</td>
<td>37585</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>36891</td>
<td>42084</td>
</tr>
<tr>
<td>CalGovernerRecall</td>
<td>0.05</td>
<td>202054</td>
<td>209112</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>199267</td>
<td>204578</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>198644</td>
<td>206481</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>259257</td>
<td>262754</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>256722</td>
<td>261865</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>251918</td>
<td>269446</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>11479</td>
<td>12517</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>10354</td>
<td>11708</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>9202</td>
<td>10818</td>
</tr>
</tbody>
</table>
### Table 5.9: The cut values for rFM and gradient rFM for $K = 256.$

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\rho$</th>
<th>repl-PaToH, rFM</th>
<th>Cut (bisection-wise)</th>
<th>repl-PaToH, rFM-gradient</th>
<th>Cut (bisection-wise)</th>
</tr>
</thead>
<tbody>
<tr>
<td>California HPN</td>
<td>0.05</td>
<td>57949</td>
<td>60868</td>
<td>57954</td>
<td>60765</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>47508</td>
<td>55299</td>
<td>48170</td>
<td>55296</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>33506</td>
<td>46024</td>
<td>33847</td>
<td>46024</td>
</tr>
<tr>
<td>Minnesota7</td>
<td>0.05</td>
<td>68613</td>
<td>86645</td>
<td>69524</td>
<td>86676</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>43859</td>
<td>71559</td>
<td>44825</td>
<td>71486</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>29059</td>
<td>55411</td>
<td>29016</td>
<td>55946</td>
</tr>
<tr>
<td>New Mexico</td>
<td>0.05</td>
<td>35863</td>
<td>52043</td>
<td>35977</td>
<td>51277</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>41882</td>
<td>48228</td>
<td>42475</td>
<td>49098</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>35483</td>
<td>33825</td>
<td>35644</td>
<td>33743</td>
</tr>
<tr>
<td>Oldenburg</td>
<td>0.05</td>
<td>42680</td>
<td>41283</td>
<td>42913</td>
<td>41344</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>39742</td>
<td>38277</td>
<td>39734</td>
<td>38204</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>35483</td>
<td>33825</td>
<td>35644</td>
<td>33743</td>
</tr>
<tr>
<td>Oregon</td>
<td>0.05</td>
<td>40339</td>
<td>55504</td>
<td>40763</td>
<td>55618</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>47350</td>
<td>59977</td>
<td>47160</td>
<td>54991</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>52547</td>
<td>59520</td>
<td>53203</td>
<td>59394</td>
</tr>
<tr>
<td>San Francisco</td>
<td>0.05</td>
<td>42435</td>
<td>71058</td>
<td>42755</td>
<td>72628</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>34045</td>
<td>55649</td>
<td>34896</td>
<td>56102</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>38604</td>
<td>47008</td>
<td>39253</td>
<td>47328</td>
</tr>
<tr>
<td>San Joaquin</td>
<td>0.05</td>
<td>61244</td>
<td>68954</td>
<td>61788</td>
<td>69460</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>46238</td>
<td>58662</td>
<td>46671</td>
<td>58638</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>28480</td>
<td>46732</td>
<td>28842</td>
<td>47063</td>
</tr>
<tr>
<td>Washington</td>
<td>0.05</td>
<td>40308</td>
<td>58844</td>
<td>41157</td>
<td>60109</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>46788</td>
<td>54376</td>
<td>47081</td>
<td>55802</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>53653</td>
<td>59881</td>
<td>52494</td>
<td>59022</td>
</tr>
<tr>
<td>Wyoming</td>
<td>0.05</td>
<td>40955</td>
<td>66101</td>
<td>41226</td>
<td>65738</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>45552</td>
<td>57917</td>
<td>46438</td>
<td>57920</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>52501</td>
<td>60785</td>
<td>52414</td>
<td>61853</td>
</tr>
<tr>
<td>CalifGovernorRecall</td>
<td>0.05</td>
<td>301908</td>
<td>309517</td>
<td>293178</td>
<td>299822</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>300748</td>
<td>315186</td>
<td>293819</td>
<td>301121</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>298685</td>
<td>308817</td>
<td>287586</td>
<td>298014</td>
</tr>
<tr>
<td>Facebook</td>
<td>0.05</td>
<td>343711</td>
<td>345023</td>
<td>355325</td>
<td>340151</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>337817</td>
<td>349196</td>
<td>338080</td>
<td>340675</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>340130</td>
<td>337885</td>
<td>327400</td>
<td>340727</td>
</tr>
<tr>
<td>Stanford</td>
<td>0.05</td>
<td>18883</td>
<td>20072</td>
<td>18869</td>
<td>20146</td>
</tr>
<tr>
<td></td>
<td>0.10</td>
<td>17022</td>
<td>18831</td>
<td>16952</td>
<td>18835</td>
</tr>
<tr>
<td></td>
<td>0.20</td>
<td>15462</td>
<td>17867</td>
<td>15452</td>
<td>17682</td>
</tr>
</tbody>
</table>
In this thesis, we proposed a heuristic based solution for the replicated hypergraph partitioning problem. In this problem, the vertices are replicated with respect to given replication amount in order to improve the quality of the partitions. This approach differs from the replication schemes in the VLSI literature in the sense that the replication of a vertex cannot bring any net to the cut, i.e., replication does not have any “side effects” except using more space. Our replication scheme can be applied to the hypergraph models in different areas such as distributed IR and spatial databases. For hypergraph partitioning, multilevel and recursive bipartitioning schemes are utilized. The basic FM heuristic is extended to a version that is capable of replication, called replicated FM which introduced new vertex states and gain update algorithms to support replication and unreplication of vertices. We adopted various well-known concepts to further improve the performance of rFM such as early-exit and gradient methodology. We proposed solutions to the issues encountered while integrating our replication scheme into the multilevel and recursive bipartitioning frameworks. These issues include removal of unnecessary replications and replica selection for nets.

The results show that replication is a valuable method to obtain partitions with better quality and the cutsize of the partitions can greatly be reduced using little amount of replication. The different properties of the hypergraphs such as average net degree have a great impact on our replication scheme as the results
indicate. Generally, our replication scheme works well with the hypergraphs with a low average net degree due to the characteristics of the FM based heuristics. Distribution of the given replication amount among bipartitionings has an important effect on the cutsize. Performing more replication at the deeper levels of the recursion tree is a better replication distribution scheme compared to distributing the given replication amount uniformly among all bipartitions. The rFM with gradient methodology can outperform rFM in certain datasets although rFM generally performs better. This may indicate that rFM with gradient methodology should be used with the hypergraphs that have relatively high average net degrees whereas rFM should be used with the hypergraphs with low average net degrees. Replication generally does not disturb the balance of the partitions. However, as the results reveal, different replication distribution schemes have certain effects on the balance of the partitions.

As future research, we have various ideas to that can further improve the quality of the partitions:

- Different operation selection strategies may be tested for rFM like allowing zero gain replication operations. Such an approach can be beneficial since excess replication may have an effect of uncovering new positive gain operations.

- We plan to try different replica selection strategies. A suitable approach may be forcing the selection of the replicas for nets after each level of the recursion tree. In this way, all replicas’ parts can be predetermined and the pins of the unused replicas can be removed.

- We can further improve the balance of the partitions by using the remaining replication amount after obtaining a $K$-way partitioning. This can be achieved by replicating vertices from the most heavily loaded part to other parts.

- A totally different approach would be using a replication scheme that operates on $K$-way partitions and using this scheme in each level of the recursion tree. In other words, we can use a $K$-way refinement heuristic that can
perform replication and unreplication operations on vertices. Performing replication in each bipartitioning has the shortcoming of having a global view over the partitions. Even if we obtain bipartitions with a cutsize value of zero, this does not guarantee that the cutsize will be zero after obtaining a $K$-way partitioning. Therefore, a $K$-way refinement heuristic can be a perfect tool to overcome this problem.
Bibliography


Replicated partitioning for undirected hypergraphs

R. Oguz Selvitopi, Ata Turk, Cevdet Aykanat
Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey

ABSTRACT

Hypergraph partitioning (HP) and replication are diverse but powerful tools that are traditionally applied separately to minimize the costs of parallel and sequential systems that access related data or process related tasks. When combined together, these two techniques have the potential of achieving significant improvements in performance of many applications. In this study, we provide an approach involving a tool that simultaneously performs replication and partitioning of the vertices of an undirected hypergraph whose vertices represent data and nets represent task dependencies among these data. In this approach, we propose an iterative-improvement-based replicated bipartitioning heuristic, which is capable of move, replication, and unreplication of vertices. In order to utilize our replicated bipartitioning heuristic in a recursive bipartitioning framework, we also propose appropriate cut-net removal, cut-net splitting, and pin selection algorithms to correctly encapsulate the two most commonly used cutsize metrics. We embed our replicated bipartitioning scheme into the state-of-the-art multilevel HP tool PaToH to provide an effective and efficient replicated HP tool, rpPaToH. The performance of the techniques proposed and the tools developed is tested over the undirected hypergraphs that model the communication costs of parallel query processing in information retrieval systems. Our experimental analysis indicates that the proposed technique provides significant improvements in the quality of the partitions, especially under low replication ratios.

Keywords: Hypergraph partitioning Recursive bipartitioning Replication Iterative improvement heuristic

1. Introduction

Models and methods based on hypergraph partitioning (HP) have been successfully used for different objectives in a wide range of areas such as parallel scientific computing [4,11,15,44], very large scale integration (VLSI) circuit layout design [1,32], parallel information retrieval (IR) [8], parallel volume rendering [9], and database systems [12,13,40].

A hypergraph is a generalization of a graph where hyperedges (nets) connect one or more vertices (cells). The HP problem is defined as the task of dividing the vertex set of a given hypergraph into disjoint subsets such that the cost (cutsize) is minimized while a certain balance constraint on the part weights is satisfied. The cutsize is generally a function of the nets that connect more than one part.

Hypergraphs can be used to represent different types of relation in a wide range of problems which can broadly be categorized into two as directed and undirected relations. Depending on the category of the relation, directed or undirected hypergraphs are used in the modeling. In undirected hypergraphs, a net is used to model an equally shared relation among the tasks/data represented by the vertices it connects. In directed hypergraphs, a net is used to model an input–output relation among the tasks/data represented by the vertices it connects.

We use the terms directional and undirectional HP models for indicating models based on partitioning of directed and undirected hypergraphs, respectively. We should note here that almost all of the state-of-the-art HP tools [2,14,26,43,45] are designed to partition undirected hypergraphs. Hence, some special techniques such as consistency condition [11] and the elementary hypergraph model [44] are utilized to model some types of directed relations correctly via undirectional HP models.

The schemes that combine vertex replication with HP models have only been studied for directional HP models in the context of VLSI circuit layout design. In these HP models, since the vertices generally model the gates or logic devices, replication corresponds to duplicating the same gate or logic device in multiple networks of a partitioned logic network. In this way, the number of connections between networks and the wiring density can be reduced at the expense of implementing the same logic in multiple networks.

In directional HP models, vertex replication may cause an increase in the cutsize, and it generally requires further replication of other vertices and nets. However, in undirectional HP models,
since an input–output relation does not exist between the vertices connected by a net, replication does not have such an effect. This forms the basic difference between vertex replication in directional and undirectional HP models. To the best of our knowledge, there are no studies in the literature addressing vertex replication schemes for undirectional HP models. In this study, we try to fill this gap. Note that, due to the above-mentioned fundamental difference in vertex replication, the techniques we present here are not directly applicable to directional HP models. Thus, replication in undirectional HP models requires specific techniques and tools tailored for this purpose.

1.1. Related work in directional HP models

Even though we do not address applications in the VLSI domain, we discuss replication schemes in this area, since, to our knowledge, VLSI circuit layout design is the only area where HP is applied together with replication, albeit in a directional partitioning framework.

Replication schemes in VLSI circuit layout design and partitioning arise in the form of gate replication to reduce pin counts and the interconnection cost of the partitioned circuits. These schemes can be categorized into two as one-phase schemes and two-phase schemes with respect to when the partitioning and replication are performed. In the one-phase approach, partitioning and replication are performed simultaneously, whereas in the two-phase approach replication is performed after obtaining a partition. In the one-phase approach, generally, extended versions of the Fiduccia–Mattheyses (FM) [17] heuristic are utilized [30, 31]. In the two-phase approach, after obtaining a partition, linear programming or flow-network [22, 23] formulations are used to achieve replication, and often, if needed, an extended FM heuristic is applied as the last step to find a feasible solution. Since this study focuses on performing replication and partitioning simultaneously, we briefly summarize the existing work on FM heuristics for directional graph/hypergraph partitioning with replication. In [30], an extended version of the FM algorithm for directional HP models is proposed to perform replication in two-way partitioned networks by introducing new definitions for cell/net states and cell gains. The authors of [31] introduce an extended version of the FM algorithm to achieve partitioning and replication, and they propose a new gain definition and objective function for this extended version. In [33], the authors use a modified FM algorithm applied over a replication graph which they obtain by a linear programming formulation. A detailed discussion and comparison of replication techniques in circuit partitioning can be found in [16].

1.2. Application

In order to show the validity of the algorithms proposed in our paper, we investigate undirectional HP models proposed for index partitioning of parallel IR systems [8, 28], where replication is beneficial and commonly used [37]. Although we address the HP models used in parallel IR, our replication scheme can be used for any domain in which the underlying problem can be modeled as an undirected hypergraph.

In parallel IR systems, the index is partitioned across several machines [7, 23, 36, 38, 41], typically in a document-based or term-based fashion, in order to process very large text collections. In [42], it is remarked that replication is necessary for improving query throughput. The authors of [35] propose a bin-packing-based greedy algorithm that utilizes query logs to distribute terms to index servers. In their experiments, they replicate a small amount of most frequent terms and discover that replication is a powerful tool in reducing the average number of per-query servers, even under low replication ratios. In the distributed IR system of Google, the entire system is replicated [5]. A selective replication scheme that replicates inverted lists of high workload terms to improve load balancing in a pipelined and term-distributed IR system is investigated in [37].

In the HP models utilized for term-based distribution of inverted indices [28], the vertex \(v_i\) represents the term \(t_i\) and the task of retrieving its inverted list. The net \(n_j\) represents the query \(q_j\) and connects the subsets of vertices that represent the terms requested by that query. In this HP model, the nets have unit costs due to the infinite result cache capacity assumption.\(^1\) The weight of a vertex is set equal to either the number of postings in the inverted list of the term represented by that vertex [8] or the multiplication of term popularity and the corresponding posting list size [37]. The balance constraint in the former vertex weighting scheme corresponds to maintaining storage balance, whereas the balance constraint in the latter vertex weighting scheme corresponds to maintaining computational workload balance. The partitioning objective of minimizing the cutsize corresponds to minimizing the communication volume during parallel query processing.

We introduce Fig. 1 to illustrate the relationship between the target application and undirectional HP models. Fig. 1(a) shows a sample term collection \(\Gamma\) that contains ten terms together with a query \(Q\) that contains six queries. Fig. 1(b) shows the undirectional hypergraph model for this sample inverted index. As seen in Fig. 1(b), net \(n_1\) connects vertices \(v_1, v_2,\) and \(v_3\), since query \(q_1\) requests the terms \(t_1, t_2,\) and \(t_3.\) Fig. 1(b) also shows a four-way partition of this hypergraph. Fig. 1(c) shows the distribution of the sample inverted index among four index servers \((I_1, I_2, I_3, I_4)\) that is induced by this four-way partition. For example, the index server \(I_3\) stores the terms \(t_3, t_4,\) and \(t_5\) and their inverted lists since part \(V_3\) of the partition consists of the vertices \(v_3, v_4,\) and \(v_5.\)

The correspondence between vertex replication and the mentioned HP model is as follows. A net in this HP model represents the undirectional shared relation among the respective retrieval tasks that can be performed concurrently and independently on the inverted lists represented by the vertices connected by that net. Thus, vertex replication corresponds to replicating inverted lists of terms for further minimization of the communication volume. For a given query, the task associated with each data is only performed by one of the processors owning the replicas of that data. Thus, the proposed scheme incurs redundant storage (data replication) but does not incur redundant computation.

1.3. Contributions

There are five main contributions of this study. (1) The differences between vertex replication in directional and undirectional HP models are explained (Section 3). (2) A vertex replication scheme for undirectional HP models is proposed (Section 4). This replication approach is based on an iterative-improvement heuristic, and it achieves replication during partitioning. For this purpose, the FM heuristic is extended to support replication and unreplication of vertices in addition to vertex moves. This extended heuristic is called rFM, and it operates on a given two-way partition (bipartition) by introducing new gain definitions and vertex states. (3) In order to utilize rFM in a recursive bipartitioning (RB) framework, appropriate cut-net removal, cut-net splitting, and pin selection algorithms are proposed to correctly encapsulate the two most commonly used cutsize metrics (Sections 5 and 6). (4) The proposed vertex replication and bipartitioning scheme is integrated into the state-of-the-art multilevel HP tool PaToH [2] that uses

---

\(^1\) This assumption simply states that each query is processed only once and its results are stored in the result cache. Further requests for the same query are responded from this result cache [10].
the RB paradigm to provide a replicated HP tool, xPaToH. Specifically, the uncoarsening phase of the multilevel framework is modified by using rFM as a replicated partitioning and refinement tool. At each level of the uncoarsening phase, the rFM algorithm is run and the multilevel scheme is extended to support replicated vertices. (5) Detailed experimental analyses are performed over the hypergraph model of the sample application (Section 1.2) using synthetic and realistic datasets. The results obtained indicate that xPaToH performs significantly better than a successful partitioning and replication scheme [28] for this application domain.

The rest of the paper is organized as follows. Section 2 gives the necessary background. Section 3 explains the differences between replication in directional and undirectional HP models. Section 4 describes the details of the rFM heuristic. Section 5 presents the proposed cut-net removal, cut-net splitting, and replication distribution schemes. Section 6 addresses the pin selection issue after obtaining a K-way partition. Section 7 discusses the results of the experiments that were carried out. Finally, Section 8 concludes.

2. Background and problem definition

2.1. Definitions and hypergraph partitioning problem

A hypergraph \( H = (V, N) \) is defined as a set of vertices \( V \) and a set of nets \( N \). Each net \( n_j \in N \) connects a subset of vertices. The set of vertices connected by net \( n_j \) is denoted as \( \text{Vertices}(n_j) \). The set of nets that connect vertex \( v_i \) is denoted as \( \text{Nets}(v_i) \). The vertices \( v_i \) and \( v_j \) are said to be neighbors if they are connected by at least one common net, i.e., \( \text{Nets}(v_i) \cap \text{Nets}(v_j) \neq \emptyset \). An \((n_i, n_j)\) tuple denotes a pair of \( n_j \) where \( v_i \in \text{Vertices}(n_j) \). The degree of a net \( n_j \) is equal to the number of vertices it connects, \( |\text{Vertices}(n_j)| \).

The total number of pins \( \mathcal{P} = \sum_{n_j \in N} |\text{Vertices}(n_j)| \) denotes the size of a given hypergraph \( H \). A weight value \( w(v_i) \) is associated with each vertex \( v_i \), and a cost value \( c(n_j) \) is associated with each net \( n_j \). The cost function for a net easily extends to a subset of nets \( \mathcal{M} \subseteq N \), i.e., \( c(\mathcal{M}) = \sum_{n_j \in \mathcal{M}} c(n_j) \).

\( \Pi = \{V_1, \ldots, V_k\} \) is a K-way partition of \( H = (V, N) \) if each part \( V_k \) is a nonempty subset of \( V \), the parts are pairwise disjoint, and the union of \( K \) parts is equal to \( V \). The weight \( W(V_k) \) of a part \( V_k \) is the sum of the weights of the vertices in that part, i.e., \( W(V_k) = \sum_{v_i \in V_k} w(v_i) \). A partition \( \Pi \) is said to be balanced if each part \( V_k \in \Pi \) satisfies the balance constraint:

\[
W(V_k) \leq (1 + \epsilon)W_{\text{avg}} \quad \text{for} \quad k = 1, \ldots, K, \tag{1}
\]

where \( W_{\text{avg}} = W(V)/K \) and \( \epsilon \) is the predetermined maximum imbalance ratio.

In a partition \( \Pi \), a net is said to connect a part if it connects at least one vertex in that part. The connectivity set \( \Lambda(n_j) \) of a net \( n_j \) is defined as the set of parts connected by \( n_j \). The number of parts in the connectivity set of \( n_j \) is denoted by \( \lambda(n_j) = |\Lambda(n_j)| \). A net is said to be cut or external if it connects more than one part \( (\lambda(n_j) > 1) \), and uncut or internal if it connects only one part \( (\lambda(n_j) = 1) \). The set of external nets in a partition \( \Pi \) is denoted as \( \mathcal{E}_\Pi \). The set of internal nets that connect a vertex \( v_i \) is denoted as InternalNets \( \{v_i\} \). Two cutsize metrics widely used in the literature to represent the cost of a partition \( \Pi \) are

\[
cutsize(\Pi) = \sum_{n_j \in \mathcal{E}_\Pi} c(n_j), \tag{2}
\]

\[
cutsize(\Pi) = (\sum_{n_j \in \mathcal{E}_\Pi} \lambda(n_j) - 1)c(n_j). \tag{3}
\]

The cost definitions in Eqs. (2) and (3) are called the cut-net metric and the connectivity metric, respectively. For example, the cut-net and connectivity metrics model the minimization of the communication volume in parallel sparse matrix vector multiplication utilizing collective and point-to-point communication schemes, respectively [11,44].

Given a hypergraph \( H = (V, N) \), hypergraph partitioning can be defined as finding a K-way partition \( \Pi = \{V_1, \ldots, V_k\} \) that minimizes the cutsize (Eqs. (2) or (3)) while maintaining the balance constraint (Eq. (1)). This problem is known to be NP-hard [32].

2.2. Iterative improvement heuristics for two-way HP

FM-based schemes [1,17] are widely used iterative-improvement heuristics to solve the HP problem. FM-based heuristics improve the cutsize of a bipartition by moving vertices from one part to the other. The gain of a vertex in these heuristics is generally defined as the reduction in the cutsize if that vertex were to be moved to its complementary part in a bipartition. FM heuristics can perform multiple passes over all vertices until the improvement in the cutsize drops below a certain threshold.

2.3. Recursive bipartitioning and multilevel frameworks

RB is the most commonly used method for obtaining a K-way partition of a hypergraph, although there are other methods based on direct K-way partitioning [3,27]. In the RB scheme, first a bipartition of the initial hypergraph is obtained, and then this
bipartition is decided to construct two subhypergraphs using the cut-net removal and cut-net splitting techniques [2] to capture the cut-net and connectivity cutsize metrics, respectively. Then these two subhypergraphs are further bipartitioned in a recursive manner. This procedure continues until desired number of parts is reached (in log \( K \) recursion levels for \( K \) parts).

FM-based heuristics perform poorly on hypergraphs with high net degrees [3,27] and small vertex degrees [19]. To alleviate these problems, multilevel algorithms have been proposed [6,20] and applied to the HP problem, leading to successful HP tools such as PaToH [2], hMeTIS [26], Mondriaan [45], Zoltan [14], and ParKWay [43].

Multilevel methodology consists of coarsening, initial partitioning, and uncoarsening phases. In the coarsening phase, the original hypergraph is coarsened into a smaller hypergraph by a sequence of coarsening levels, where, in each level, various matching and clustering algorithms are used to form super-vertices from highly coherent vertices. Coherent vertices are the vertices that share high number of nets. In the initial partitioning phase, a bipartition of the coarsest hypergraph is obtained, and this coarsest hypergraph is projected back to the original hypergraph in the uncoarsening phase. At each level of the uncoarsening phase, FM-based or KL-based [29] refinement heuristics are used to improve the quality of the bipartitions.

3. Replication in directional versus undirectional HP models

There are two main differences between vertex replication in directional and undirectional HP models. (i) The replication of a vertex in directional HP models may bring internal nets to the cut and thus can increase the cutsize of a partition, and (ii) vertex replication generally requires further net and pin replication in directional HP models.

In directional hypergraphs, the nets that connect a vertex \( v_i \) are categorized as input and output nets of \( v_i \). In a dual manner, the vertices that are connected by a net \( n_j \) are categorized as input and output vertices of \( n_j \). For example, in hypergraph representation of gate-level VLSI circuits for layout design [1] and column-net hypergraph representation of sparse matrices for parallel matrix–vector multiplication [11], nets have single input and multiple output vertices, which correspond to vertices having multiple input and single output nets.

In directional HP models, when an output vertex \( v_i \) of an internal net \( n_j \) is replicated, \( n_j \) becomes cut since any new instance of the replicated vertex \( v_i' \) must be fed by \( n_j \) on the part it is replicated to. Fig. 2 shows an example of vertex replication in a directed hypergraph. A sample bipartition on this directed hypergraph is illustrated in Fig. 2(a). Initially, the cutsize of the bipartition is one, assuming that the nets have unit costs. As shown in Fig. 2(b), when \( v_3 \) is replicated, \( n_2 \) and \( n_3 \) become cut since \( v_3' \) has to be fed by both of these nets, pins \((n_2, v_3')\) and \((n_3, v_3')\) generated in Fig. 2(b). Furthermore, when an external net \( n_k \)’s input vertex \( v_i \) is replicated, \( n_k \) is generally replicated together with \( v_i \) to be able to save \( n_k \) from the cut. As shown in Fig. 2(b), when \( v_1 \) is replicated, \( n_3 \) is also replicated, leading to the addition of a new net \( n_3' \) and a new pin \((n_3, v_3')\) in \( V_h \). In this way, we are able to save \( n_3 \) from the cut. However, since \( n_2 \) and \( n_3' \) become cut, the cutsize of the bipartition increases from one to two after the replication.

In contrast, in undirectional HP models, performing replication does not bring internal nets to the cut, and putting additional pins to the new instances of the replicated vertices may not be necessary, since a net represents a shared relation rather than a dependence among the vertices it connects. In other words, we can make a choice among the instances of a replicated vertex for a net in order to decide which one of these instances will represent that replicated vertex. This is done by putting a pin only to a single instance of the replicated vertex for that net. Fig. 3 shows an example of vertex replication in an undirected hypergraph. The initial bipartition is seen in Fig. 3(a), which is the undirected version of the directed hypergraph in Fig. 2(a) and has a cutsize of one. As opposed to replication of \( v_3 \) in Fig. 2, replication of \( v_3 \) in Fig. 3 does not bring any internal net to the cut, since, as seen in Fig. 3(b), the nets \( n_1 \) and \( n_2 \) are not required to feed \( v_3' \). Instead, \( n_1 \) (or similarly \( n_2 \) and \( n_3 \)) can ‘choose’ to use either \( v_2 \) or \( v_3' \), since \( n_1 \) just needs to select an instance for this replicated vertex. In other words, \( n_1 \) has to have just one pin to an instance of the replicated vertex, which is selected to be the pin \((n_1, v_2)\) in this example. We refer to this problem as the pin selection problem and address it in Section 6. After replication of \( v_3 \), the cutsize of the bipartition reduces from one to zero.

Having described the differences between vertex replication in directional and undirectional HP models, we set our focus on replication in undirectional HP models and define the **Replicated Undirected Hypergraph Partitioning** problem as follows: given an undirected hypergraph \( H = (\mathcal{V}, E) \), an imbalance ratio \( \epsilon \), and a replication ratio \( \rho \), find a \( K \)-way covering subset of \( \mathcal{V} \). \( P^* = \{V_1, V_2, \ldots, V_K\} \) that minimizes the cutsize (Eqs. (2) or (3)) while satisfying the following constraints.

- **Balancing constraint**: \( W_{\text{max}} \leq (1 + \epsilon)W_{\text{avg}} \), where \( W_{\text{max}} = \max_{1 \leq k \leq K} W(V_k) \) and \( W_{\text{avg}} = (1 + \rho)W(V)/K \).
- **Replication constraint**: \( \sum_{k=1}^{K} W(V_k) \leq (1 + \rho)W(V) \).
Note that $W_{\text{max}}$ denotes the weight of the maximally weighted part, $W_{\text{avg}}$ denotes the part weight under perfect balance, and $W(V)$ denotes the total vertex weight without replication.

### 4. Replicated FM (rFM)

We propose an extended FM heuristic which we call replicated FM (rFM) to address the Replicated Undirected Hypergraph Partitioning problem.

#### 4.1. Definitions

In a two-way covering subset $\mathcal{I}^R = \{A,B\}$ of $V$, a vertex can belong to $V_A$, $V_B$, or both of them if it is replicated, and hence it can be in one of three states, $A$, $B$, and $AB$:

$$\text{State } (v_i) = \begin{cases} 
A & \text{if } v_i \in V_A \text{ and } v_i \not\in V_B, \\
B & \text{if } v_i \in V_B \text{ and } v_i \not\in V_A, \\
AB & \text{if } v_i \in V_A \text{ and } v_i \in V_B.
\end{cases}$$

Herein, a covering subset $\mathcal{I}^R$ of $V$ will be referred to as a replicated partition of $V$, and subsets of $\mathcal{I}^R$ will be referred to as parts of $\mathcal{I}^R$. Each instance of a replicated vertex is referred to as a replica. The number of non-replicated vertices in state $A$ and connected by $n_j$ is denoted as $\sigma_A(n_j)$. The number of non-replicated vertices in state $B$ and connected by $n_j$ is denoted as $\sigma_B(n_j)$. Similarly, the number of replicated vertices (not the number of replicas) that are connected by $n_j$ is denoted as $\sigma_{AB}(n_j)$. Note that, according to the definitions,

$$|\text{Vertices}(n_j)| = \sigma_A(n_j) + \sigma_B(n_j) + \sigma_{AB}(n_j).$$

A net $n_j$ in a two-way replicated partition is said to be cut if both $\sigma_A(n_j) > 0$ and $\sigma_B(n_j) > 0$. The cut-state of a net is used to describe whether that net is cut or not. A net $n_j$ is said to be internal to $V_A$ if $\sigma_A(n_j) = 0$ and it is said to be internal to $V_B$ if $\sigma_B(n_j) = 0$. A net $n_j$ can be considered internal to either $V_A$ or $V_B$ if $\sigma_A(n_j) = 0$, $\sigma_B(n_j) = 0$ and $\sigma_{AB}(n_j) > 0$.

rFM is an iterative-improvement heuristic that tries to improve the cutsize of a given two-way replicated partition by move, replication, and unreplication operations performed on vertices. The move and replication operations can only be performed on non-replicated vertices, whereas the unreplication operation can only be performed on replicated vertices. A non-replicated vertex has two gains, which are move and replication gains. Similarly, a replicated vertex also has two gains, which are unreplication from $V_A$ and unreplication from $V_B$ gains. The gain definitions are as follows.

- The **move gain**, $g_m(v_i)$, of a non-replicated vertex $v_i$ is defined as the reduction in the cutsize if vertex $v_i$ were to be moved to the other part. The move gain of $v_i$ is equal to the difference between the sum of the costs of the nets saved from the cut and the sum of the costs of the internal nets that are brought to the cut. Fig. 4(b) and (a) display the move of $v_1$ from $V_A$ to $V_B$. Moving $v_1$ from $V_A$ to $V_B$ brings net $n_1$ into the cut while saving net $n_2$ from the cut. Hence, $g_m(v_1) = c(n_2) - c(n_1)$. After the move operation, $v_1$ is locked. The locked vertices in the examples are illustrated by gray color.

- The **replication gain**, $g_r(v_i)$, is defined as the reduction in the cutsize if vertex $v_i$ were to be replicated to the other part. The replication gain of $v_i$ is equal to the sum of the costs of the nets saved from the cut. When a vertex is replicated, it cannot bring any internal net to the cut and thus cannot increase the cutsize. This forms the basic difference between the move and replication operations. Consequently, for any vertex $v_i$, we have $g_m(v_i) > 0$ and $g_r(v_i) > 0$. Fig. 4(b) and (c) show the replication of $v_1$ from $V_A$ to $V_B$. The replication of $v_1$ saves net $n_1$ from the cut as the move of $v_1$ does; however, net $n_2$ still remains as an internal net, as opposed to the move operation on the same vertex. Hence, $g_r(v_1) = c(n_2)$. In the examples, if a net is internal to a part and connects a replicated vertex, we illustrate this by putting a pin to the replica that is in the part of the internal net and omit the pin to the other replica. In contrast, if an external net connects a replicated vertex, the pins to the replicas of the replicated vertex connected by that net are displayed by dashed lines.

- The **unreplication gain**, $g_u(v_i)$, is defined as the reduction in the cutsize if a replica of the replicated vertex $v_i$ were to be unreplicated from its part. Since unreplication of a replica cannot improve the cutsize, the maximum unreplication gain of a replica is zero. Thus, for any replicated vertex $v_i$, $g_u(A,v_i) \leq 0$ and $g_u(B,v_i) \leq 0$. A replica with an unreplication gain of zero implies that this replica is unnecessary and its removal will not change the cutsize. On the other hand, if the unreplication gain of a replica is negative, this implies that the replica is necessary and its unreplication will bring internal net(s) to the cut. Fig. 5 shows the unreplication of a necessary and an unnecessary replica. Initially, there are two replicas of $v_1$ in the bipartition in Fig. 5(b). The replica in $V_A$ is necessary, and its unreplication causes the internal net $n_1$ to be cut, as seen in Fig. 5(a). On the other hand, the replica in $V_B$ is unnecessary, and its unreplication does not change the cut set, as seen in Fig. 5(c). Hence, $g_u(A,v_1) = -c(n_1)$ and $g_u(B,v_1) = 0$.

#### 4.2. Overall rFM algorithm

Replated FM performs a predetermined number of passes considered on all vertices, where each pass comprises a sequence of operations (Algorithm 1). First, we compute the two possible gains for each vertex and initialize the pin distributions of the nets (line 1). At the beginning of each pass, we unlock all vertices to be able to perform operations on them (line 3). Then the algorithm enters the inner while loop (lines 4–7). In this loop, we first select a vertex and an operation (move, replication, or unreplication) to be performed on the selected vertex (line 5) according to the operation selection criteria described below. Then we perform the selected operation if it does not violate the size constraints on the weights of the parts (line 6). After the selected operation is performed on the vertex, the selected vertex is locked and the gain values of its unlocked neighbors and the pin distributions of the nets that connect this vertex are updated (line 7). A pass terminates when there are no more valid operations. At the end of a pass, a
rollback procedure is applied to the point where the partition with the minimum cutsize is seen (line 8). The size constraint check performed during the operation selection is done as follows. (i) If the selected operation is a move or a replication, the new weight of the destination part if the selected operation were to be performed is computed, and, if it exceeds \((1 + \epsilon) W_{\text{avg}}\), this operation is discarded, and (ii) if the selected operation is unreplication, it is checked if the weight of the part on which unreplication were to be performed drops below \((1 - \epsilon) W_{\text{avg}}\) and, if it does, it is discarded. Furthermore, if the selected operation is replication, it is only performed if the total amount of replication performed up to that point plus the weight of the selected vertex does not exceed the allowed replication amount \(\rho W(V)\).

**Algorithm 1**: Basic steps of rFM.

\[
\text{Input: } \mathcal{H} = (V, \mathcal{A}), \mathcal{P} = \{V_A, V_B\} \\
1 \text{ Initialize pin distributions, gains, and priority queues.} \\
2 \textbf{while} there are passes to perform \textbf{do} \\
3 \quad Unlock all vertices. \\
4 \quad while there is any valid operation \textbf{do} \\
5 \quad \quad (v, \text{op}) \leftarrow \text{Select the vertex and the operation to perform on it.} \\
6 \quad \quad \text{Perform op on } v, \text{ store the reduction in the cutsize, and lock } v. \\
7 \quad \quad Update the gains of unlocked neighbors of } v \text{ and the pin distributions of the nets in } \text{Nets}(v). \\
8 \quad \text{Rollback to the point when minimum cutsize is seen.}
\]

**Operation selection**: We use a priority-based selection approach for determining the current operation and disallow some operations that do not satisfy certain conditions. The selection strategy is based on principles such as minimizing the number of unnecessary replicas, limiting the replication amount, and improving the balance. We give the highest priority to the elimination of unnecessary replicas. We do not perform unreplication operations with negative gains simply because such operations will degrade the cutsize. If there are no unnecessary replicas, we make a choice between move and replication by selecting the operation with the higher gain. Ties between the gains of the selected move and replication operations are broken in favor of the move operations. Any replication with a gain value of zero is disabled because such operations will produce unnecessary replicas. However, the zero-gain moves that improve the balance are retained. Since, for any vertex \(v_i\), \(g_i(v_i) \geq g_0(v_i)\), in a single pass, the number of replication operations tends to outweigh the number of move operations. This issue can be addressed by the gradient methodology, which we discuss below.

**Gradient methodology**: The gradient methodology is used in FM heuristics that are capable of replication for directed graph models [34] to obtain partitions with better cutsize. The basic idea of the gradient methodology is to introduce the replication in the later iterations of a pass, especially when the improvement achieved in the cutsize by performing only move operations drops below a certain threshold. As mentioned in [16], early replication can have a negative effect on the final partition by limiting the algorithm’s ability to change the current partition. Furthermore, by using the replication in the later iterations, the algorithm can climb out of the local minima reached by the move operations. In rFM, we adopt and modify the gradient methodology by allowing only move and unreplication operations until the improvement in the cutsize drops below a certain threshold, and then we allow replication operations.

**Early exit**: We use the early-exit scheme [18] to improve the run-time performance of rFM. In this scheme, if there are no improvements in the cutsize for a predetermined number of successive iterations, the current pass of the FM algorithm is terminated since it is unlikely to further improve the cutsize.

**Locking**: In conventional move-based FM algorithms, after moving a vertex, it is locked to avoid thrashing [17]. Similarly, in rFM, we also lock the operated vertex after performing a move, replication, or unreplication operation on that vertex.

**Data structures**: We maintain six priority queues keyed according to the gain values of the vertices with respect to type of operation. The heaps are implemented as binary heaps. For each part, we have three heaps for storing the move, replication, and unreplication gains. The two gains associated with a non-replicated vertex are stored in the move and replication heaps of the part that the vertex belongs to. Similarly, the two gains associated with the replicas of a replicated vertex have their unreplication gains stored in the unreplication heap of their respective parts.

### 4.3. Net criticality

The main power of rFM, like all FM-based algorithms, lies in its efficient linear-time gain update operations [17]. In this section, we present net criticality definitions that trigger updates on move, replication, and unreplication gains.

A net \(n_j\) is said to be critical to part \(V_k\), if an operation performed on a vertex \(v_i \in V_k\) can change the cut-state of \(n_j\). Whenever an operation is performed on a vertex \(v_i\), we check the criticality conditions of the nets that connect \(v_i\). If the criticality condition of a net \(n_j\) that connects \(v_i\) changes, the other vertices that are connected by \(n_j\) are checked for gain updates. Each type of operation imposes different pin distributions for the criticality of nets; thus the criticality definition of a net is classified as move criticality, replication criticality, and unreplication criticality, according to the type of operation that causes a change in the cut-state of the respective net.

For a net to be move critical, it must connect at least two non-replicated vertices \((\sigma_m(n_j) + \sigma_d(n_j) > 1)\), and it must either be an internal net or an external net with a single pin in one of the two parts. As seen in Table 1, a net \(n_j\) is move critical to \(V_k\) if \((\sigma_m(n_j) = 1)\) and \((\sigma_d(n_j) > 0)\) or \((\sigma_m(n_j) > 0)\) and \((\sigma_d(n_j) > 1)\), and to \(V_{\bar{k}}\) if \((\sigma_m(n_j) = 0)\) and \((\sigma_d(n_j) > 1)\) or \((\sigma_m(n_j) > 1)\) and \((\sigma_d(n_j) > 0)\).

For a net to be replication critical, it must connect at least two non-replicated vertices \((\sigma_r(n_j) + \sigma_d(n_j) > 1)\), and it must be an
external net with a single pin in one of the two parts. As seen in Table 1, a net $n_i$ is replication critical to $V_A$ if $(\sigma_A(n_i) = 1$ and $\sigma_B(n_i) > 0)$, and to $V_B$ if $(\sigma_B(n_i) = 1$ and $\sigma_A(n_i) > 0)$. Note that the internal nets which are always move critical are never replication critical, since the replication of a vertex connected by an internal net cannot change the cut-state of that net. This difference is indicated in Table 1, where the conditions $(\sigma_A(n_i) = 0$ and $\sigma_B(n_i) > 1)$ and $(\sigma_A(n_i) = 0$ and $\sigma_B(n_i) > 1)$, which exist in the move-critical column, do not appear in the replication-critical column.

For a net to be unreplication critical, it must be an internal net that connects at least one non-replicated and one replicated vertex $(\sigma_A(n_j) + \sigma_B(n_j) > 0$ and $\sigma_{AB}(n_j) > 0)$. As seen in Table 1, a net $n_j$ is unreplication critical to $V_A$ if $(\sigma_A(n_j) = 0$ and $\sigma_B(n_j) > 0$ and $\sigma_{AB}(n_j) > 0)$, and to $V_B$ if $(\sigma_B(n_j) = 0$ and $\sigma_A(n_j) > 0$ and $\sigma_{AB}(n_j) > 0)$. Note that external nets that connect a single non-replicated vertex in only one of the two parts, which are move critical, are never unreplication critical, since unreplication of a vertex connected by an external net cannot change the cut-state of that net. This difference is indicated in Table 1, where the conditions $(\sigma_A(n_j) = 1$ and $\sigma_B(n_j) > 0$) and $(\sigma_B(n_j) = 1$ and $\sigma_A(n_j) > 0)$, which are shown in the move-critical column do not appear in the unreplication-critical column.

### 4.4. rFM algorithm details

In this section, we present detailed explanations of some of the non-trivial concepts and algorithms used in rFM. The examples respect the basics of the operation selection criteria mentioned in Section 4.2. For the sake of simplicity, we assume that each net has unit cost, and we also overlook the balance constraints on part weights in the examples.

**Initial gain computation.** The initial gain computation, which is performed at the beginning of each pass of rFM, is given in Algorithm 2 and consists of two main loops. The first loop resets the initial gain values by traversing vertices (lines 1–7) and the second loop completes the initialization of gains by traversing all pins (lines 8–18). The move and replication gains are computed according to the external and critical nets that connect these vertices, whereas the unreplication gains are modified according to the internal and critical nets that connect these vertices.

The move and replication gains of the non-replicated vertices are initially set to their minimum possible values (lines 3–4). If a net $n_i$ is external and move or critical or replication critical, the move and replication gains of the vertices connected by $n_i$ must be incremented by $c(n_i)$ (lines 12–13), since it can be saved from the cut with either one of these operations. In contrast to move and replication gains, unreplication gains are initially set to their maximum possible values (lines 6–7). If a net $n_i$ is internal and thus unreplication critical, the unreplication gains of the replicas of the replicated vertices connected by $n_i$ may need to be updated. The unreplication gains of the replicas that are in the same part with this internal net need to be decremented by $c(n_i)$ if $n_i$ connects at least one non-replicated vertex that is in the same part with this net (lines 14–18).

**Algorithm 2: Initial move, replication, and unreplication gain computation.**

| Input: $N = (V_A, V_B), P^R = \{V_A, V_B\}$
| 1 foreach $v_i \in V_A$ do
| 2 if $State(v_i) \neq AB$ then
| 3 $g_m(v_i) \leftarrow -c(InternalNets(v_i))$
| 4 $g_r(v_i) \leftarrow 0$
| 5 else
| 6 $g_m(v_i) \leftarrow 0$
| 7 $g_r(v_i) \leftarrow 0$
| 8 foreach $n_i \notin N$ do
| 9 foreach $v_i \in Vertices(n_i)$ do
| 10 if $State(v_i) \neq AB$ and $n_i$ is external then
| 11 if $(\sigma_A(n_i) = 1$ State(v_i) = A) or $(\sigma_B(n_i) = 1$ State(v_i) = B) then $n_i$ is critical to $V_A$ or $V_B$
| 12 $g_m(v_i) \leftarrow g_m(v_i) + \sigma(n_i)$
| 13 $g_r(v_i) \leftarrow g_r(v_i) + c(n_i)$
| 14 else if $State(v_i) = AB$ and $n_i$ is internal then
| 15 if $\sigma_A(n_i) > 0$ and $\sigma_B(n_i) = 0$ then $n_i$ is critical to $V_A$
| 16 $g_m(v_i) \leftarrow g_m(v_i) - c(n_i)$
| 17 else if $\sigma_B(n_i) > 0$ and $\sigma_A(n_i) = 0$ then $n_i$ is critical to $V_B$
| 18 $g_m(v_i) \leftarrow g_m(v_i) - c(n_i)$

Fig. 6(a) shows the pin distributions of the nets and the gain values of the vertices for a sample bipartition after Algorithm 2 is run on this sample. Nets $n_4$, $n_5$, and $n_6$ are cut; thus the cutsize of the bipartition in Fig. 6(a) is three. We use the notation $\sigma(n_i) = (\sigma_A(n_i) : \sigma_B(n_i) : \sigma_{AB}(n_i))$ to denote the pin distribution of $n_i$.

**Gain updates after a move operation.** Algorithm 3 shows the procedure for performing gain updates after moving a given vertex $v^*$ from $V_A$ to $V_B$. The algorithm includes updating fields of $v^*$ (lines 1–2), the pin distributions of Nets ($v^*$) (lines 4 and 16), and the gain values of neighbors of $v^*$ (lines 5–15 and 17–27). The necessary field updates on $v^*$ are performed by updating the state and locked fields of $v^*$ to reflect the move operation. The pin distribution of each net $n_i \in Nets(v^*)$ needs to be updated by decrementing $\sigma_A(n_i)$ by 1 and incrementing $\sigma_B(n_i)$ by 1. When the pin distribution of $n_i$ changes, its criticality may change with respect to the operation type. The change in the criticality of $n_i$ may require various gain updates on the unlocked vertices connected by $n_i$.

After decrementing the number of vertices of $n_i$ in $V_A$ (line 4), we check the value of $\sigma_B(n_i)$ to see if the criticality of $n_i$ has changed (lines 5 and 11). If $\sigma_B(n_i) = 0$, then $n_i$ becomes internal to $V_B$ by becoming move critical and unreplication critical to this part, and if $\sigma_A(n_i) = 1$, then $n_i$ becomes move critical and replication critical to $V_A$. Similarly, after incrementing the number of vertices connected by $n_i$ in $V_B$ (line 16), we check the value of $\sigma_A(n_i)$ to see if the criticality of $n_i$ has changed (lines 17 and 23). If $\sigma_A(n_i) = 1$, then $n_i$ was internal and hence was move critical and replication critical to $V_B$. Under these conditions for $n_i$, the gains of the vertices connected by $n_i$ should be checked for any update with respect to the corresponding part.
In Fig. 6(a), when we consider the selection criteria, the selected operation is going to be the move of $v_4$ whose gain is one. Fig. 6(b) shows the bipartition after running Algorithm 3 with the selected vertex $v_3$. After the move of $v_4$, $n_5$ is saved from the cut, and the cutsize of the bipartition becomes two.

**Gain updates after a replication operation.** Algorithm 4 shows the procedure for performing gain updates after replicating a given vertex $v^*$ from $V_A$ to $V_B$. The procedure starts with changing the state of $v^*$ to $AB$ and locking both replicas of $v^*$ (lines 1–2). Then, for each net $n_i$ that connects $v^*$, the pin distributions of $n_i$ are updated and checked for criticality condition changes (lines 6 and 17). Since $v^*$ was in $V_A$, before replication, $\sigma_{AB}(n_i)$ is decremented by 1 and $\sigma_{AB}(n_i)$ is incremented by 1 to reflect that $v^*$ is now a replicated vertex (lines 4–5). The replication of $v^*$ from $V_A$ does not change the $\sigma_{AB}(n_i)$ value of any $n_i \in$ Nets ($v^*$); thus the criticality conditions that include $\sigma_{AB}(n_i)$ need not be checked.

After the value of $\sigma_{AB}(n_i)$ is decremented (line 4), $n_i$ must be checked for criticality condition changes to see if there are any necessary gain updates for the neighbors of $v^*$ (lines 6 and 17). If $\sigma_B(n_i) = 0$, $n_i$ becomes move critical and unreplication critical to $V_B$. In this condition, the move gains of the unlocked vertices and the unreplication gains of the unlocked replicas that are connected by $n_i$ need to be decremented by $c(n_i)$ since $n_i$ is internal now, and the move of any vertex or the unreplication of any replica connected by $n_i$ would bring it to cut. If $\sigma_A(n_i) = 1$, $n_i$ becomes move critical and replication critical to $V_A$. The move or the replication of the only non-replicated vertex $v_i$ connected by $n_i$ in $V_A$ can now save $n_i$ from the cut, and thus the move and replication gains of this vertex must be incremented by $c(n_i)$.

After moving $v_4$, now we are to select another vertex to operate on in Fig. 6(b). There are two operations with the highest gain, which are the replication of $v_5$ and the replication of $v_6$, and the gain values of these operations are one. We select to replicate $v_6$. Fig. 6(c) shows the bipartition after running Algorithm 4 with $v_6$.

After replication of $v_6$, we observe that $n_4$ is now uncut, and the cutsize becomes one.

**Gain updates after an unreplication operation.** Algorithm 5 shows the procedure for performing updates after unreplication of a given replica $v^*$ from $V_A$. The procedure starts with changing the state of $v^*$ to $B$ and locking it (lines 1–2). Then, for each net $n_i$ that connects $v^*$, the pin distributions of $n_i$ are updated and checked for criticality condition changes (lines 6 and 17). Since $v^*$ was a replicated vertex before unreplication from $V_A$, $\sigma_B(n_i)$ is incremented by 1 and $\sigma_{AB}(n_i)$ is decremented by 1 to reflect that $v^*$ is now a non-replicated vertex in $V_B$ (lines 4–5). The unreplication of $v^*$ from $V_B$ does not change the $\sigma_{AB}(n_i)$ value of any $n_i \in$ Nets ($v^*$); thus the criticality conditions that include $\sigma_{AB}(n_i)$ need not be checked.

After the value of $\sigma_{AB}(n_i)$ is incremented (line 4), $n_i$ must be checked for criticality condition changes to see if there are any necessary gain updates for the neighbors of $v^*$ (lines 6 and 17). If $\sigma_A(n_i) = 1$, it means that $n_i$ was move critical and unreplication critical to $V_A$. In this case, the move and replication gains of the unlocked vertices and replicas that are in $V_A$ and connected by $n_i$ are incremented by $c(n_i)$, since $n_i$ is not an internal net anymore.
If $\sigma B(n_i) = 2$, it means that $n_i$ was move critical and replication critical to $V_B$. The net $n_i$ connects two vertices in $V_B$ and one of them, $v^*$, is already locked, and thus the move and replication gains of the other vertex, $v$, need to be decremented by $c(n_i)$, since this vertex can no longer save $n_i$ from the cut.

In Fig. 6(c), after the replication of $v^*$, there is an unnecessary replica in $V_B$ with an unreplication gain of zero. According to the selection criteria, the selected operation is the unreplication of the replica of $v_1$ in $V_B$. Fig. 6(d) shows the bipartition after running Algorithm 5. The unreplication of an unnecessary replica cannot change the cutsize; thus, after the unreplication of the replica $v_1$ in $V_B$, the cutsize is still one.

### Algorithm 3: Gain updates after moving $v^*$ from $V_A$ to $V_B$.

Input: $H = (V, N, \Pi^R = \{V_A, V_B\})$, $v^* \in V_A$

1. State($v^*$) ← $B$
2. Lock $v^*$
3. foreach $n_i \in$ Nets($v^*$) do
   4. $\sigma A(n_i) ← \sigma A(n_i) - 1$
   5. if $\sigma A(n_i) = 0$ then $n_i$ becomes critical to $V_B$
   6. if State($v_i$) = $B$ then
      7. $g_m(v_i) ← g_m(v_i) - c(n_i)$
      8. else if State($v_i$) = $AB$ then
         9. $g_m(v_i) ← g_m(v_i) - c(n_i)$
      10. else if $\sigma A(n_i) = 1$ then $n_i$ becomes critical to $V_A$
      11. if State($v_i$) = $A$ then
         12. $g_m(v_i) ← g_m(v_i) + c(n_i)$
         13. $g_r(v_i) ← g_r(v_i) + c(n_i)$
         14. if State($v_i$) = $B$ then
            15. $g_m(v_i) ← g_m(v_i) + c(n_i)$
            16. if $\sigma A(n_i) = 1$ then $n_i$ becomes critical to $V_B$
            17. if State($v_i$) = $A$ then
               18. $g_m(v_i) ← g_m(v_i) + c(n_i)$
               19. $g_r(v_i) ← g_r(v_i) + c(n_i)$
               20. else if State($v_i$) = $AB$ then
                  21. $g_m(v_i) ← g_m(v_i) + c(n_i)$
                  22. else if $\sigma A(n_i) = 2$ then $n_i$ becomes critical to $V_B$
                  23. if State($v_i$) = $B$ then
                     24. $g_m(v_i) ← g_m(v_i) - c(n_i)$
                     25. $g_r(v_i) ← g_r(v_i) - c(n_i)$

### Algorithm 5: Gain updates after unreplicating $v^*$ from $V_A$.

Input: $H = (V, N, \Pi^R = \{V_A, V_B\})$, $v^* \in V_A$

1. State($v^*$) ← $B$
2. Lock $v^*$
3. foreach $n_i \in$ Nets($v^*$) do
   4. $\sigma A(n_i) ← \sigma A(n_i) - 1$
   5. if $\sigma A(n_i) = 0$ then $n_i$ becomes critical to $V_B$
   6. if $\sigma B(n_i) = 1$ then $n_i$ was critical to $V_A$
   7. if State($v_i$) = $A$ then
      8. $g_m(v_i) ← g_m(v_i) + c(n_i)$
      9. if $\sigma A(n_i) = 1$ then $n_i$ became critical to $V_A$
      10. else if State($v_i$) = $AB$ then
          11. $g_m(v_i) ← g_m(v_i) - c(n_i)$
          12. else if $\sigma A(n_i) > 0$ then
              13. $g_m(v_i) ← g_m(v_i) - c(n_i)$
          14. else if $\sigma A(n_i) = 2$ then $n_i$ became critical to $V_B$
          15. if State($v_i$) = $B$ then
             16. $g_m(v_i) ← g_m(v_i) - c(n_i)$
             17. if $\sigma A(n_i) > 0$ then
                18. $g_m(v_i) ← g_m(v_i) - c(n_i)$

### 4.5. Complexity analysis of rFM

Consider a single pass of rFM to be performed on an initial bipartition $\Pi^R = \{V_A, V_B\}$ of a hypergraph $H = (V, N)$ with $V = |V|$ vertices and $P$ pins. Let $V_A$ be the number of replicated vertices and $V_B$ be the number of non-replicated vertices. Clearly, $V = V_A + V_B$. The initial gain computation takes $O(P)$ time since the vertices connected by each net are traversed as seen in Algorithm 2. After the initial gain computation is completed, these gain values are stored in six heaps. For each heap, it is required to perform a build-heap operation. The build-heap operations on two heaps storing move gains take a total of $O(V_A)$ time. Similarly, the build-heap operations on two heaps storing replication gains take a total of $O(V_B)$ time. This is because the total number of vertices in two heaps storing move gains and in two heaps storing replication gains are both equal to $V$. The build-heap operation on the heap storing unreplication gains of the replicas in $V_A$ takes $O(V_B)$ time, and similarly the build-heap operation on the heap storing unreplication gains of the replicas in $V_B$ takes $O(V_A)$ time, since each heap possesses $V_A$ elements. Thus, the total time required for building heaps is equal to $O(V_A + V_B + V_A + V_B) = O(2V) = O(V)$.

The selection procedure consists of checking maximum gain values in six heaps, which takes $O(1)$ time. After selecting the gain value from one of the heaps with respect to the selection criteria, we perform an extract-max operation on the selected heap and a delete operation on another heap for the other gain value of the selected vertex (Section 4.2). Regardless of the selected heap, the extract-max and delete operations on the heaps are bounded by the number of total vertices, since the maximum number of elements in a single heap can be at most $V$. Thus, a single selection operation takes $O(1) + O(2 \log V) = O(\log V)$ time. In a single pass of rFM where all vertices are exhausted, we can make at most $V$ selections. Consequently, the cost of selection in a single pass of rFM is equal to $O(V \log V)$.

As proved in the original FM heuristic [17], during an FM pass, the criticality state of a net changes at most three times due to the vertex locking mechanism adopted, which limits the number of gain updates by a constant factor. For our algorithm, Table 1 reveals that the criticality of a net $n_i$ depends on its pin distributions, $\sigma A(n_i)$ and $\sigma B(n_i)$. More specifically, after an operation is performed, the
criticality of $n_i$ changes if $\sigma_A(n_i)$ or $\sigma_B(n_i)$ observes the following changes:

- $\sigma_A(n_i)$ from $2 \rightarrow 1$ or $1 \rightarrow 0$ or $0 \rightarrow 1$ or $1 \rightarrow 2$.
- $\sigma_B(n_i)$ from $2 \rightarrow 1$ or $1 \rightarrow 0$ or $0 \rightarrow 1$ or $1 \rightarrow 2$.

Consider a vertex $v_i$ connected by a net $n_i$. Recall that move of $v_i$ from $V_A$ to $V_B$ will decrease $\sigma_A(n_i)$ by one and increase $\sigma_B(n_i)$ by one. Similarly, replication of $v_i$ from $V_A$ to $V_B$ will decrease $\sigma_A(n_i)$ by one and unreplication of $v_i$ from $V_A$ will increase $\sigma_A(n_i)$ by one. Since replication of $v_i$ will decrease $\sigma_A(n_i)$ or $\sigma_B(n_i)$ by one, replication of vertices connected by $n_i$ can change the cut state of that net at most four times. Note that, when there are two vertices connected by $n_i$ that are locked in two parts of a bipartition, that net will always be cut during that pass, and hence further criticality analysis will not be necessary for it. Also observe that a move to a part locks the moved vertex to that part and unreplication of a vertex from a part locks the other replica of that vertex in the other part. It is easy to see that the number of criticality changes that can be achieved by any combination of move and unreplication operations is at most five (the criticality of $n_i$ can change at most four times for one part and one time for the other part). If we add criticality changes that can be achieved by replication as well, we can surmise that the criticality state of a net can change at most nine times during a pass. In fact, a more careful analysis reveals that the criticality state of a net can change at most seven times, since some changes either overlap with each other or cancel each other in the sense that, when one of them occurs for a net, the other cannot. Thus, in rFM, the number of criticality state changes for a net, and hence the number of gain updates for vertices is also bounded by a constant factor.

In rFM, since each vertex possesses two gains, in the worst case, both of these gain values may need to be updated, which doubles the number of gain updates for any vertex compared to FM. Each gain update requires an increase-key or decrease-key operation on the corresponding heap. Consequently, the complexity of gain updates in rFM is $O(2P \log V) = O(P \log V)$.

Given these complexity values, the complexity of a single pass of rFM is $O(P + V + V \log V + P \log V) = O(P \log V)$, since $P \geq V$.

4.6. rFM and multilevel framework

We utilize the rFM heuristic in a multilevel framework. In our multilevel approach we use the coarsening and initial partitioning phases of the conventional multilevel approach as is, and redesign the uncoarsening phase from scratch so that we can perform replication in this phase.

At each level of the uncoarsening phase, we perform multiple phases to refine the current bipartition. At the end of each level $l$, the bipartition $\Pi_l^R$ on the coarser hypergraph $H_l$ is projected back to the bipartition $\Pi_{l-1}^R$ on the finer hypergraph $H_{l-1}$. The projection includes the decomposition of each super-vertex in $H_l$ to its constituent vertices in $H_{l-1}$. The decomposition of a non-replicated super-vertex in $H_l$ results in multiple non-replicated vertices in $H_{l-1}$. Similarly, the decomposition of a replicated super-vertex in $H_l$ results in multiple replicated vertices in $H_{l-1}$. The existence of replicated vertices does not disturb the projection process. Clearly, the decomposition of a replicated super-vertex to its constituent replicated vertices will not change the cut-state of the nets this replicated super-vertex is connected by.

Unnecessary replicas tend to occur excessively at the beginning of each uncoarsening level due to the increase in the degrees of freedom after the projection of a coarser hypergraph to a finer hypergraph. Such replicas hamper the quality of the refinement and partitioning process if they are not removed, since (i) they consume the gain replication amount needless, which may prevent the positive gain replications from being performed, and (ii) in the construction of the new hypergraphs for further levels of the RB, they can cause the new hypergraphs to become unnecessarily bigger. In operation selection, we give the unreplication of unnecessary replicas the highest priority (Section 4.2). This way, the majority of the unnecessary replications are eliminated at the beginning of each uncoarsening level.

5. Recursive bipartitioning

To obtain $K$-way replicated partitions, we utilize rFM in a recursive bipartitioning framework, which requires cut-net removal and cut-net splitting techniques to be altered to support replication. In the RB framework, after each bipartition of $H = (V, A)$, two subhypergraphs $H' = (V', A')$ and $H'' = (V'', A'')$ are constructed from $\Pi^R = \{V_A, V_B\}$. In replicated or non-replicated HP, regardless of the underlying cost scheme (e.g., cut-net, connectivity), the vertex sets of $H'$ and $H''$ are equivalent to $V_A$ and $V_B$, respectively. That is, $V = V_A$ and $V'' = V_B$.

Recall that, in replicated HP, $V_A$ and $V_B$ include the replicated vertices by definition (Section 4.1). In the construction of $H'$ and $H''$, the replicas of the replicated vertices of $\Pi^R$ become non-replicated vertices of both $H'$ and $H''$, and the necessary pins are placed for these vertices. Consequently, the numbers of vertices and pins of the resulting hypergraphs are greater when compared to non-replicated HP.

In the following two subsections, for RB-based replicated HP, we show how to extend the cut-net removal and cut-net splitting schemes to cut the cut-net (Eq. (2)) and connectivity (Eq. (3)) cutoff metrics, respectively.

5.1. Cut-net removal

In the cut-net removal scheme for RB-based non-replicated HP, the internal net sets of $V_A$ and $V_B$ constitute the net sets of $H'$ and $H''$, respectively. Vertices connected by those internal nets will again be connected by the same nets in the new subhypergraphs. All cut-nets are discarded since they contribute to the cutoff only once.

In the cut-net removal scheme for RB-based replicated HP, the cut-nets are also discarded, since the definition of a cut-net in replicated HP does not take replicated vertices into account (a net $n_i$ is cut if $\sigma_A(n_i) > 0$ and $\sigma_B(n_i) > 0$). Internal nets are kept for further bipartitionings as in non-replicated HP. The net sets of $H'$ and $H''$ are defined as follows:

$$N' = \{ n'_j \text{ with Vertices}(n'_j) = \text{Vertices}(n_j) \text{ s.t.} \quad n_j \in N \text{ and } \sigma_A(n_j) = 0 \}.$$  

$$N'' = \{ n''_j \text{ with Vertices}(n''_j) = \text{Vertices}(n_j) \text{ s.t.} \quad n_j \in N, \sigma_A(n_j) = 0 \text{ and } \sigma_B(n_j) > 0 \}.$$  

In the construction of the new net sets, there is a subtle difference (indicated with $\sigma_B(n_j) > 0$) due to the nets that connect only replicated vertices. In this case, since such nets may be considered to be internal either to $V_A$ or $V_B$, we assumed them to be internal to $V_A$ and added the necessary pins accordingly and discarded the pins to the replica in $V_B$. However, a more intelligent scheme can be devised to decide on which part these nets are considered to be internal. Fig. 7(a) shows an example of a cut-net removal scheme, where the cut-net $n_k$ is removed while the internal nets $n_i$ and $n_j$ are preserved as $n'_i$ and $n''_j$ for further bipartitionings.

5.2. Cut-net splitting

In the cut-net splitting scheme for RB-based non-replicated HP, the internal net sets of $V_A$ and $V_B$ will be in the net sets of $H'$.
and $\mathcal{H}'$, respectively. Furthermore, each cut-net $n_j \in \mathcal{N}'$ is split into two new nets, $n'_j$ and $n''_j$, where $\text{Vertices}(n'_j) = \text{Vertices}(n_j) \cap \mathcal{V}_A$ and $\text{Vertices}(n''_j) = \text{Vertices}(n_j) \cap \mathcal{V}_B$. Then, $n'_j$ is added to $\mathcal{N}'$ if $\sigma_A(n'_j) > 1$ and $n''_j$ is added to the $\mathcal{N}''$ if $\sigma_B(n''_j) > 1$. Clearly, $\sigma_A(n'_j) = 0$ and $\sigma_A(n''_j) = 0$.

In the cut-net splitting scheme for RB-based replicated HP, we need to add pins to the replicas of the replicated vertices in order to preserve the flexibility of performing move or replication operations on them in the newly constructed hypergraphs. The internal nets are kept for further bipartitionings as in non-replicated HP. The net sets of $\mathcal{H}'$ and $\mathcal{H}''$ are defined as follows:

$$\mathcal{N}' = \left\{ n'_j \text{ with } \text{Vertices}(n'_j) = \text{Vertices}(n_j) \cap \mathcal{V}_A \text{ s.t. } n_j \in \mathcal{N}, \sigma_A(n_j) > 1 \right\}.$$ 

$$\mathcal{N}'' = \left\{ n''_j \text{ with } \text{Vertices}(n''_j) = \text{Vertices}(n_j) \cap \mathcal{V}_B \text{ s.t. } n_j \in \mathcal{N}, \sigma_B(n_j) > 1 \right\}.$$ 

Notice that the new net definitions encompass both the internal nets and the external nets that are split. Similar to the cut-net removal scheme, there is a subtle difference (indicated with $\sigma_B(n_j) > 1$) due to the nets that connect only replicated vertices. Such nets are handled in the same way as they are handled in cut-net removal scheme. Fig. 7(b) shows the splitting of $n_j$ into two distinct nets $n'_j$ and $n''_j$ and the addition of necessary pins for these nets for further bipartitionings. Note that a pin is added for each of the replicas of the replicated vertex $v_i$.

5.3 Replication amount distribution

The RB scheme consists of multiple bipartitions. The replication amount used in each bipartitioning can have an effect on the final cutsize. We consider two different replication amount distribution schemes in this work. (i) In the level-wise replication scheme, first, the total replication amount is divided by $\log K$, the number of levels of the recursion tree of RB, and then, for each specific level, the assigned replication amount is evenly distributed among the bipartitions in that level. (ii) In the bisection-wise replication scheme, the total replication amount is divided by $K - 1$, the number of bipartitionings in a $K$-way partitioning, and then distributed evenly among these bipartitionings.

6. Pin selection

After achieving a $K$-way replicated partition, in order to compute the cutsize we have to select the pins to the replicas of the replicated vertices connected by a net. The replication of a vertex $v_i$ brings the problem of selecting replicas of $v_i$ for each net it is connected by. If a net $n_j$ connects replicated vertices, we need to decide which replicas of these replicated vertices will be “used” by $n_j$. This is required for a couple of reasons: (i) the cutsize of the final partition can only be computed after deciding from which part the replicas will be used, and (ii) the investigated real-world problem may enforce the nets to make a choice from which parts their replicas will be used. We propose a simple pin selection technique whose basic motivation is not to increase the cutsize with careless pin selection.

Fig. 8 shows two pin selection alternatives for a net $n_j$ which connects three vertices $v_1$, $v_2$, and $v_3$ in a three-way partition given in Fig. 8(a). The vertices $v_1$, $v_2$, and $v_3$ are replicated, each having three replicas, and $v_3$ is non-replicated. In the examples for pin selection, after a selection is performed for a pin, this pin is indicated by a thick line. A selection alternative for $n_j$ is seen in Fig. 8(b) where $v_1$, $v_2$, and $v_3$ are selected from $V_1$ and $V_2$, respectively and $\lambda(n_j) = 3$. A more careful selection alternative is shown in Fig. 8(c), where both $v_1$ and $v_2$ are selected from $V_2$ and $\lambda(n_j) = 1$. This example illustrates how pin selection can be crucial in computing the cutsize of a given partition.

Let $nr(n_j, k)$ and $r(n_j, k)$ respectively denote the number non-replicated and replicated vertices that are connected by $n_j$ in $V_k$. Consider a net $n_j$ that connects a replicated vertex $v_i$ which has $r$ replicas. We are to make a decision for $n_j$ to select one of these replicas within the considerations mentioned above. Our replica selection algorithm is based on a greedy heuristic that consists of two stages.

The first stage of the algorithm is based on the following observation. Consider a cut-net $n_j$ that connects at least one non-replicated vertex in $V_k$ (i.e., $nr(n_j, k) > 0$). If $r(n_j, k) > 0$ too, then the pins of $n_j$ to the replicas in $V_k$ can be safely selected.
without degrading the cutsize. Using this observation, for each net \( n_j \), the algorithm traverses \( n_j \)'s pins to the vertices in \( \Lambda(n_j) \) and selects the currently unselected pins to the replicas in \( V_k \) if \( nr(n_j, k) > 0 \). If a pin \((n_j, v^k_i)\) is selected for the replicated vertex \( v_i \) during this process, all other pins \((n_j, v^k_j)\), where \( \ell \neq k \), are deselected. Here, \( v^k_i \) denotes the replica of the replicated vertex \( v_i \) in \( V_k \). After the selection of pin \((n_j, v^k_i)\) for \( n_j \), the \( r(n_j, k) \) value is decremented by one for each \( V_k \) that a replica of \( v_i \) resides in. At the end of the first stage, for each net \( n_j \), \( n_j \)'s pins to the non-replicated vertices are selected by default, since these are the only candidates for those pins. After the selection of pin \((n_j, v_i)\) for \( n_j \), where \( v_i \in V_k \) is non-replicated, the \( nr(n_j, k) \) value is decremented by one. \Fig{9} illustrates our selection heuristic. In \Fig{9}(a), there are two non-replicated \((v_a, v_b)\) and four replicated \((v_a, v_b, v_t, v_u)\) pins. \Fig{9}(b) shows the resulting pins after the first stage of the pin selection algorithm is run. The selected pins \((v_a, v_b), (v_t, v_u), (v_a, v_b), (v_a, v_b), (v_t, v_u)\) are shown with thick lines. Note that deselected pins \((v_a, v_b), (v_t, v_u)\) are removed.

At the beginning of the second stage of the algorithm, all \( nr(n_j, k) \) values are equal to zero. For a net \( n_j \), if \( r(n_j, k) > 0 \) for at least one \( V_k \), the pin selection problem for \( n_j \) can be reduced to the set cover problem. This case can be seen in \Fig{9}(b) for the pin selection process for \( n_j \). In the example, our sets are \( S_1 = \{v_r, v_s, v_t\} \) and \( S_2 = \{v_r, v_t\} \), and we try to find a set cover of the ground set \( S = \{v_r, v_s, v_t\} \). Since the set cover problem is NP-hard [25], we adopt a simple greedy heuristic that has an approximation ratio of \( 1 + \ln(n) \) [24], where \( n \) is the number of total elements. Basically, in each iteration, this greedy heuristic selects the set that covers the largest number of uncovered elements so far and then removes the currently covered elements from remaining sets. When this algorithm is run for \( n_j \) in the example in \Fig{9}(b), \( S_1 \) will be selected as the covering subset which is illustrated in \Fig{9}(c). According to this selection process, the pins \((n_j, v^k_1), (n_j, v^k_2), (n_j, v^k_3), (n_j, v^k_4)\) are selected whereas the pins \((n_j, v^k_5), (n_j, v^k_6), (n_j, v^k_7)\) are deselected.

7. Experiments
7.1. Experimental setup

The proposed replication scheme is integrated into the multilevel HP tool PaToH [2]. We call this modified version of PaToH rpPaToH. In the experiments, we used the same parameters for PaToH and rpPaToH in the coarsening and the initialization partitioning phases. We used agglomerative clustering (absorption clustering using pins) and greedy hypergraph growing algorithms in the coarsening and the initialization partitioning phases, respectively. For both PaToH and rpPaToH, the imbalance ratio is set to \( \epsilon = 0.10 \) in all experiments. The boundary FM (BFM) refinement heuristic option is selected for PaToH, whereas the proposed rFM heuristic is used for rpPaToH. The number of passes for the refinement algorithms used is set to three for both tools. For the early-exit feature, the number of allowed operations which do not improve the cutsize is set to 100 for both PaToH and rpPaToH. We report results for five different \( K \) (16, 32, 64, 128, 256) and five different \( \rho \) (0.05, 0.10, 0.15, 0.20, 0.25) values. The results for \( K = 16 \) are omitted in Tables 3 and 5 due to lack of space.

All algorithms are implemented in C and compiled in gcc with the -O3 flag. Due to the randomized nature of PaToH, all of the partitioning results reported are the averages of ten runs. In the experiments, a six-core AMD Opteron with 2.1 GHz of clock frequency and 32 GB of main memory is used.

The performance of the proposed replication scheme and the replicated HP tool rpPaToH developed is evaluated on term-based partitioning of inverted indices for the parallel query processing application discussed in Section 1.2. The datasets are separated into two as realistic and semi-synthetic datasets. The realistic dataset used in the experiments is the AOL dataset [39], which consists of about 12 million queries and 1.3 million terms. We used synthetic datasets due to the difficulties in obtaining real-world query sets for IR. The synthetic datasets are in fact semi-synthetic in the sense that they are generated from real-world crawls downloaded from the Stanford WebBase Project [21]. The CG dataset is composed of pages collected from sites related to the California governor election on 09/30/2003. The FB dataset is composed of pages collected from Facebook on 09/08/2008. The WP dataset is composed of pages collected from Wikipedia in May 2006. The VG dataset is composed of pages collected from sites related with the Virginia Tech shooting on 04/23/2007. We generated query sets from each dataset consisting of queries between two to six terms, since 90% of Web search queries have between one and six terms [39]. Queries with single terms are not included in the datasets since they induce nets with single pins, and such nets do not incur any cost to the cutsize. In order to imitate real-world query sets, documents are randomly selected from these datasets, and then, from these selected documents, query terms are formed so that the query term frequencies follow a Zipfian distribution. Utilizing these query sets, we generated hypergraphs as described in [28] except the vertex weighting schema. Since vertex replication does not incur redundant computation due to the pin selection scheme described
7.2. Performance evaluations

In this section, we provide a thorough performance analysis of PaToH in terms of imbalance, replication utilization, and cutsize improvement.

Table 3 compares the performances of four different schemes utilized in PaToH in terms of cutsize averages for all datasets normalized with respect to those of the bis + nor scheme.

Table 4 provides data for the performance evaluation of PaToH+ MF and rpPaToH.

In Section 6, the balance constraint is interpreted as balancing the number of terms assigned to each processor by using unit vertex weights in these hypergraphs. The characteristics of the hypergraphs generated from these datasets are given in Table 2.

In parallel IR, one of the most commonly used replicated partitioning schemes is based on first hash-based partitioning of the inverted index and then replication of the posting lists of the most frequently occurring terms across all parts [37]. We call this scheme Hash+ MF. The recently proposed HP-based index partitioning scheme, which is summarized in Section 1.2, is reported to reduce the parallel query processing overhead significantly compared to hash-based partitioning. So, we replaced the hash-based partitioning in Hash+ MF with PaToH to obtain a more effective replicated inverted index partitioning scheme, which is referred here as PaToH+ MF. In our implementation...
Table 5
cutsize \((\times 10^3)\) values for PaToH+MF and rpPaToH.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>(\rho)</th>
<th>(K = 32) PaToH+MF</th>
<th>K = 64 PaToH+MF</th>
<th>(\rho)</th>
<th>(K = 128) PaToH+MF</th>
<th>(\rho)</th>
<th>(K = 256) PaToH+MF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>0.05</td>
<td>359</td>
<td>220</td>
<td>0.10</td>
<td>330</td>
<td>206</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>601</td>
<td>287</td>
<td>0.10</td>
<td>539</td>
<td>267</td>
<td>0.15</td>
</tr>
<tr>
<td>VG</td>
<td>0.05</td>
<td>657</td>
<td>355</td>
<td>0.10</td>
<td>495</td>
<td>308</td>
<td>0.15</td>
</tr>
<tr>
<td>WP</td>
<td>0.05</td>
<td>685</td>
<td>490</td>
<td>0.10</td>
<td>512</td>
<td>306</td>
<td>0.15</td>
</tr>
<tr>
<td>FB</td>
<td>0.05</td>
<td>3774</td>
<td>1254</td>
<td>0.10</td>
<td>2161</td>
<td>780</td>
<td>0.15</td>
</tr>
<tr>
<td>AOL</td>
<td>0.05</td>
<td>359</td>
<td>220</td>
<td>0.10</td>
<td>330</td>
<td>206</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Averages of normalized cutsize values of PaToH+MF and rpPaToH with respect to PaToH+MF:

Table 6 displays the averages of the percentage reduction in cutsize values of PaToH+MF and rpPaToH.

<table>
<thead>
<tr>
<th>(\rho)</th>
<th>(K = 32) PaToH+MF</th>
<th>(\rho)</th>
<th>(K = 64) PaToH+MF</th>
<th>(\rho)</th>
<th>(K = 128) PaToH+MF</th>
<th>(\rho)</th>
<th>(K = 256) PaToH+MF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>1.00</td>
<td>0.54</td>
<td>1.00</td>
<td>0.60</td>
<td>1.00</td>
<td>0.66</td>
<td>1.00</td>
</tr>
<tr>
<td>0.10</td>
<td>1.00</td>
<td>0.54</td>
<td>1.00</td>
<td>0.60</td>
<td>1.00</td>
<td>0.67</td>
<td>1.00</td>
</tr>
<tr>
<td>0.15</td>
<td>1.00</td>
<td>0.53</td>
<td>1.00</td>
<td>0.60</td>
<td>1.00</td>
<td>0.66</td>
<td>1.00</td>
</tr>
<tr>
<td>0.20</td>
<td>1.00</td>
<td>0.51</td>
<td>1.00</td>
<td>0.58</td>
<td>1.00</td>
<td>0.65</td>
<td>1.00</td>
</tr>
<tr>
<td>0.25</td>
<td>1.00</td>
<td>0.49</td>
<td>1.00</td>
<td>0.56</td>
<td>1.00</td>
<td>0.63</td>
<td>1.00</td>
</tr>
</tbody>
</table>

As seen in the table, both algorithms provide replicated partitions within the allowed imbalance values. The balancing performance of both algorithms is comparable, and there is no clear winner in this performance metric. Apart from that, it can be said that, as the given replication amount increases, the balance of the partitions obtained gets better. This is because replication can also be used to improve the balance of the partitions obtained.

The replication utilization values for the PaToH+MF replication scheme are not presented in Table 3 since PaToH+MF always utilizes 100% of the given replication amount. As seen in Table 3, rpPaToH does not fully utilize the given replication amount. This is because (i) replication operations with zero gain value are not allowed, which may prevent the replication operations from being performed even though the cutsize may be greater than zero. This is because each of the partitioning obtained. (ii) the gradient methodology uses more move operations, which limits the amount of replication performed in a pass of rFM. The remaining replication amount can be utilized by a post-processing step to rpPaToH such as MF or any other replication scheme to further improve the cutsize. However, the main purpose of the experiments in this section is to test the validity of RB-based replicated HP. Hence, the results with such post-processing enhancements are not reported here.

Table 7 displays the cutover values obtained using PaToH+MF and rpPaToH. Without any exceptions, rpPaToH performs significantly better than PaToH+MF in all experiments. In both schemes, as expected, the cutover value decreases with increasing replication amount. The bottom of Table 7 displays the averages of normalized cutover values of PaToH+MF and rpPaToH with respect to those of PaToH+MF. As K increases for a fixed \(\rho\), the average performance gap between rpPaToH and PaToH+MF decreases. However, even for the highest K value of 256, rpPaToH reduces the cutover by 29%, 29%, 31%, and 32% for the \(\rho\) values 0.05, 0.10, 0.15, 0.20, and 0.25, respectively. As \(\rho\) increases for a fixed K, the average performance gap between rpPaToH and PaToH+MF increases gradually.
Table 6 The averages of the percentage reduction in cutsize values of all datasets for rpPaToH over PaToH.

<table>
<thead>
<tr>
<th>ρ</th>
<th>K = 16</th>
<th>K = 32</th>
<th>K = 64</th>
<th>K = 128</th>
<th>K = 256</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>76.76</td>
<td>68.28</td>
<td>60.69</td>
<td>52.47</td>
<td>44.64</td>
</tr>
<tr>
<td>0.10</td>
<td>81.01</td>
<td>72.75</td>
<td>65.02</td>
<td>56.87</td>
<td>49.36</td>
</tr>
<tr>
<td>0.15</td>
<td>83.98</td>
<td>75.80</td>
<td>68.30</td>
<td>60.32</td>
<td>53.11</td>
</tr>
<tr>
<td>0.20</td>
<td>86.19</td>
<td>78.43</td>
<td>71.02</td>
<td>63.06</td>
<td>55.95</td>
</tr>
<tr>
<td>0.25</td>
<td>87.69</td>
<td>80.55</td>
<td>73.42</td>
<td>65.42</td>
<td>58.37</td>
</tr>
</tbody>
</table>

PaToH decreases with increasing K. This experimental finding can be attributed to the fact that, with increasing K, the cutsize increases, and thus the ratio of the reduction in the cutsize due to fixed replication decreases.

Table 7 shows the run-time averages of all datasets for rpPaToH normalized with respect to those of PaToH. For a fixed K value, the run-time performance of rpPaToH degrades with respect to PaToH as ρ increases, since replication introduces new vertices and pins during the partitioning process. For a fixed ρ value, the run-time difference between PaToH and rpPaToH increases in favor of PaToH for increasing K. Our analysis reveals that this is mainly due to the run-time differences in the coarsening and the initial partitioning phases of PaToH and rpPaToH. Note that, after each bipartitioning in rpPaToH, the further coarsening and initial partitioning phases generally have to work on larger hypergraphs than those in PaToH. The larger the K, the greater the number of times these larger hypergraphs have to be bipartitioned, and hence the difference between PaToH and rpPaToH grows with increasing K. However, even for the largest ρ = 0.25 and K = 256 values, rpPaToH is only 2.78 times slower than PaToH on the average.

8. Conclusions and future work

A vertex replication scheme is proposed for undirectional HP models. The proposed scheme achieves replication during the partitioning process. Replication is performed using an extended version of the FM iterative-improvement heuristic (rFM) that operates on two-way partitions and is capable of replication.
and unreplication in addition to move of vertices. This two-way replicated partitioning scheme is used in a recursive bipartitioning framework to obtain $K$-way replicated partitions. Regarding the replicated vertices in a $K$-way replicated partition, a simple pin selection algorithm is proposed for the nets that connect replicated vertices. We developed a multilevel replicated HP tool, referred to as $rpPaToH$, by embedding our replication scheme into the uncoarsening phase of the multilevel HP tool $PaToH$.

The validity of the proposed replication scheme is tested on one realistic and four semi-synthetic information retrieval datasets. $rpPaToH$ is compared with a state-of-the-art replication scheme that replicates the most frequent terms to all parts to show that $rpPaToH$ achieves better improvements in the cutsize within the allowed imbalance values with relatively low replication utilization. This work shows that vertex replication can be very effective in reducing the cutsize of the partitions obtained using HP by using little amount of replication.

As future research, we consider various ideas that can further improve the quality of the partitions. (i) Different operation selection strategies can be tested for rFM such as allowing zero gain replication or negative gain unreplication operations. (ii) The remaining replication amount can be used in a clever $K$-way replication heuristic to further improve the cutsize. Alternative ways of distributing the given replication amount between $rpPaToH$ and this $K$-way replication heuristic can further be investigated.

References


R. Oguz Selvitopi received his M.Sc. degree in Computer Engineering from Bilkent University, Turkey, in 2010. He is currently a Ph.D. candidate at Bilkent University. His research interests are parallel and distributed systems, scientific computing, and algorithms.

Ata Turk received his B.Sc. and M.Sc. degrees from the Computer Engineering Department of Bilkent University, Turkey, in 2002 and 2004, respectively. He is currently working towards a Ph.D. degree at Bilkent University. His research interests include parallel information retrieval and algorithms.

Cevdet Aykanat received his B.S. and M.S. degrees from Middle East Technical University, Ankara, Turkey, both in Electrical Engineering, and his Ph.D. degree from Ohio State University, Columbus, US, in Electrical and Computer Engineering. He was a Fulbright scholar during his Ph.D. studies. He worked at the Intel Supercomputer Systems Division, Beaverton, Oregon, US, as a research associate. Since 1989, he has been affiliated with the Department of Computer Engineering, Bilkent University, Ankara, Turkey, where he is currently a professor. His research interests mainly include parallel computing, parallel scientific computing and its combinatorial aspects, parallel computer graphics applications, parallel data mining, graph and hypergraph theoretic models for load balancing, high performance information retrieval systems, parallel and distributed databases, and grid computing. He has (co)authored about 60 technical papers published in academic journals indexed in the ISI, and his publications have received about 500 citations in ISI indexes. He is the recipient of the 1995 Young Investigator Award of The Scientific and Technological Research Council of Turkey and 2007 Parlar Science Award. He was appointed a member of IFIP Working Group 10.3 (Concurrent System Technology) in April 2004, a member of the EU-INTAS Council of Scientists in June 2005, and an Associate Editor of IEEE Transactions of Parallel and Distributed Systems in December 2008.
BALANCE PRESERVING MIN-CUT REPLICATION SET FOR A K-WAY HYPERGRAPH PARTITIONING

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Volkan Yazıcı
September, 2010
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Özcan Öztürk

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Oya-Ekin Karaşan

Approved for the Institute of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Institute
ABSTRACT

BALANCE PRESERVING MIN-CUT REPLICATION SET FOR A K-WAY HYPERGRAPH PARTITIONING

Volkan Yazıcı
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2010

Replication is a widely used technique in information retrieval and database systems for providing fault-tolerance and reducing parallelization and processing costs. Combinatorial models based on hypergraph partitioning are proposed for various problems arising in information retrieval and database systems. We consider the possibility of using vertex replication to improve the quality of hypergraph partitioning. In this study, we focus on the Balance Preserving Min-Cut Replication Set (BPMCRS) problem, where we are initially given a maximum replication capacity and a K-way hypergraph partition with an initial imbalance ratio. The objective in the BPMCRS problem is finding optimal vertex replication sets for each part of the given partition such that the initial cutsize of the partition is improved as much as possible and the initial imbalance is either preserved or reduced under the given replication capacity constraint. In order to address the BPMCRS problem, we propose a model based on a unique blend of coarsening and integer linear programming (ILP) schemes. This coarsening algorithm is based on the Dulmage-Mendelsohn decomposition. Experiments show that the ILP formulation coupled with the Dulmage-Mendelsohn decomposition-based coarsening provides high quality results in feasible execution times for reducing the cost of a given K-way hypergraph partition.

Keywords: partitioning, hypergraph partitioning, replication.
ÖZET

K PARÇALI BIR HİPERÇİZGE BÖLÜMLEMESİ İÇİN
DENGE KORUMALI MİN-KESİT ÇOKLAMA KÜMESİ

Volkan Yazıcı
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2010


Anahtar sözcükler: bölümleme, hiperçizge bölümleme, çoklama.
Acknowledgement

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Cevdet Aykanat for the continuous support of my M.S. study and research, for his patience, motivation, enthusiasm, and immense knowledge.

Besides my advisor, I would like to thank the rest of my thesis committee: Asst. Prof. Dr. Özcan Öztürk and Assoc. Prof. Dr. Oya-Ekin Karaşan, for their encouragement, insightful comments, and hard questions.

I also would like to thank Ata Türk, Enver Kayaarslan, Tayfun Küçükyılmaz, Reha Oğuz Selvitopi, Önder Bulut, and Ahmet Camcı, for their valuable contributions, fruitful hints and inspiring discussions.
Anneme, Babama ve Kardeşime...
## Contents

1 Introduction 1

2 Preliminaries 5

2.1 K-Way Hypergraph Partitioning 5

2.2 K-Way Hypergraph Partitioning With Vertex Replication 7

2.3 The Dulmage-Mendelsohn Decomposition 8

3 Balance Preserving Min-Cut Replication Set 12

3.1 Boundary Adjacency Hypergraph Construction 13

3.2 Vertex Selection in Boundary Adjacency Hypergraph 18

3.3 Coarsening of Boundary Adjacency Hypergraph 20

3.4 Balance Preserving Replication Capacity Computation 23

3.5 Part Visit Order for Replication 24

4 Experimental Results 25

4.1 Data Set Collection 25
CONTENTS

4.2 Implementation Details ........................................ 27
4.3 Initial $K$-Way Hypergraph Partitioning ....................... 28
4.4 Replication Results ........................................... 28
4.5 Comparison of Coarsening Algorithms and The Effect of Coarsening 32
4.6 Part Visit Orderings .......................................... 33
4.7 System Resource Consumptions ................................ 34

5 Conclusion ......................................................... 35

A SUK to MCRS Transformation .................................... 36

B Finding the Cutsize of a Partition With Vertex Replications 38
List of Figures

2.1 The Dulmage-Mendelsohn decomposition. ..................... 10

3.1 A 3-way partition of a sample hypergraph $\mathcal{H}$. .................. 16

3.2 Sample boundary adjacency hypergraph construction. ............... 16

3.3 Sample net splitting problem. .................................. 17

3.4 The fine-grained Dulmage-Mendelsohn decomposition of sample boundary adjacency hypergraph $\mathcal{H}_i^\text{con}$. ......................... 22

A.1 Sample SUK problem to MCRS problem transformation. ............. 37
List of Tables

4.1 Data set properties........................................ 27
4.2 Properties of hypergraph partitions.................... 29
4.3 Replication results........................................ 31
Chapter 1

Introduction

In the literature, combinatorial models based on hypergraph partitioning are proposed for various complex and irregular problems arising in parallel scientific computing [4, 16, 17, 27, 65, 66], VLSI design [2, 41, 46], information retrieval [15], software engineering [8], and database design [25, 26, 43, 47, 62]. These models formulate an original problem as a hypergraph partitioning problem, trying to optimize a certain objective function (e.g., minimizing the total volume of communication in parallel volume rendering, optimizing the placement of circuitry on a die area, minimizing the access to disk pages in processing GIS queries) while maintaining a constraint (e.g., balancing the computational load in a parallel system, using disk page capacities as an upper bound in data allocation) imposed by the problem. In general, the solution quality of the hypergraph partitioning problem directly relates to the formulated problem. Hence, efficient and effective hypergraph partitioning algorithms are important for many applications.

Combinatorial models based on hypergraph partitioning can broadly be categorized into two groups. In the former group, which we call as undirectional hypergraph partitioning models, hypergraphs are used to model a shared relation among the tasks or data represented by the vertices. For instance, hypergraph partitioning models used in database design, information retrieval [15], and GIS queries [25, 26] can be categorized in this group. In the latter group, which we call as directional hypergraph partitioning models, hypergraphs are used to model a
directional (source-destination) relation among the tasks or data represented by the vertices. For example, hypergraph partitioning models used in matrix vector multiplication [18, 19, 70] and VLSI design [2, 41, 46] can be categorized in this group. In this study, we focus on the undirectional hypergraph partitioning models. Directional hypergraph partitioning models are out of the scope of this work.

Replication is a widely used technique in information retrieval and database systems. This technique is generally used for providing fault-tolerance (e.g., maximizing the availability of data in case of a disk failure) and reducing parallelization (e.g., minimizing communication costs in information retrieval systems) and processing (e.g., minimizing disk access costs of a database system) costs. We consider the possibility of using vertex replication to improve the quality of partitioning objective in undirectional hypergraph models. We refer to this problem as hypergraph partitioning with vertex replication and there are two viable approaches to this problem. In the first approach, which we call as one-phase, replication is performed concurrently with the partitioning. A concurrent work proposes a heuristic for this problem in [60]. In the second approach, which we call as two-phase, replication is performed in two separate phases: In the first phase, hypergraph is partitioned and in the second phase, replication is applied to the partition produced in the previous phase. In this study, we propose an efficient and effective replication phase based on a unique blend of an integer linear programming (ILP) formulation and a coarsening algorithm. This coarsening algorithm is based on the Dulmage-Mendelsohn decomposition. In this approach, we iterate over available parts and try to find replication sets corresponding to the vertices that are to be replicated into iterated parts. Replication set of each part is constrained by a maximum replication capacity. Replication sets should be determined in such a way that the partition imbalance is preserved after the replication.

In the literature, there are various studies for replication in different domains. Below we discuss related studies from VLSI design, relational and spatial databases, and information retrieval domains.
CHAPTER 1. INTRODUCTION

In VLSI design, the first in-depth discussion about logic replication is given by [56], where they propose a heuristic approach. Later, [44] and [51] extend the Fiduccia and Mattheyses (FM) iterative improvement algorithm to allow vertices to be duplicated during partitioning. [37] proposes a network flow model to the optimal replication for min-cut partitioning, and an FM based heuristic to the size constrained min-cut replication problem. [45] introduces the concept of functional replication. [69] provides an optimal solution to the min-area min-cut replication problem. [3] presents a survey about circuit partitioning and provide a brief list of existing logic replication schemes. [31] provides enhancements for available gate replication heuristics.

Replication is a well-studied topic in database literature as well, and it is generally coupled with reliability, fault recovery, and parallelization. There are various publications about distributed databases and replication dating back to mid-70s [22, 40, 61]. A majority of these studies are concerned about fault recovery and thus apply full replication of the whole database. [53] presents a survey of current state of the art technologies in distributed database systems. As noted by [33], database systems encapsulate major implications within itself (e.g. transaction management [7], recoverability [9, 23], and serializability [7]) and considering the dynamic nature of the databases, studied methodologies in distributed database replication mainly focus on the consistency issues. With the impact of geographical information systems in the past decade, there has been a growing interest in the storage modelling of large-scale spatial network databases [25, 26] and multidimensional access methods [32, 58]. [63] provides models for declustering and load-balancing in parallel geographic information systems. [59] gives a survey of data partitioning and replication management in distributed geographical information systems. [57] provides a survey of replicated declustering schemes for spatial data. [35] presents a selective data replication scheme for distributed geographical data sets.

Another application area where replication is dubbed as indispensable is search and information retrieval systems. There are many surveys [5, 34, 50, 67, 68] investigating the fundamental concepts of the field. With the growing need for performance and wide acceptance of distributed computing, traditional
information retrieval concepts are augmented for scalability, parallelization and fault-tolerance purposes. Caching, clustering and replication concepts are utilized to enhance these architectures. [36] proposes a text retrieval system which utilizes clustering and full replication of the data structures for scalability purposes. [48] gives a comparison of replication and caching approaches for information retrieval systems. [6] presents an overview of the clustering architecture deployed at Google, where they exploit replication via sharding through clusters. [12, 13, 14] present distributed information retrieval architectures utilizing different clustering and replication models. They present their findings on the effects of networking and query distribution over the performance of replication and clustering. [49] proposes a pipelined distributed information retrieval model, where a naive partial replication scheme duplicating the most frequent terms on all disks.
Chapter 2

Preliminaries

In this chapter, notations that will be used throughout the thesis and the Dulmage-Mendelsohn decomposition is given. In Section 2.1, $K$-way hypergraph partitioning is presented. Next, in Section 2.2, partitioning with vertex replication is given. Finally, in Section 2.3, a brief explanation of the Dulmage-Mendelsohn decomposition will be shown.

2.1 $K$-Way Hypergraph Partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a two-tuple, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{N}$ denotes the set of nets (hyperedges) among those vertices. Every net $n \in \mathcal{N}$ connects a subset of vertices. The vertices connected by a net $n$ are called its pins and denoted as $\text{Pins}(n) \subseteq \mathcal{V}$. Two vertices are said to be adjacent if they are connected by at least one common net. That is, $v \in \text{Adj}(u)$ if there exists a net $n$ such that $u, v \in \text{Pins}(n)$. A weight $w(v)$ and a cost $c(n)$ are assigned for each vertex $v$ and net $n$, respectively. Adjacency $\text{Adj}(\cdot)$ and weight $w(\cdot)$ operators easily extend to a set $U$ of vertices, that is, $\text{Adj}(U) = \left( \bigcup_{u \in U} \text{Adj}(u) \right) - U$ and $w(U) = \sum_{v \in U} w(v)$.

A $K$-way vertex partition of $\mathcal{H}$ is denoted as $\Pi(\mathcal{V}) = \{V_1, V_2, \ldots, V_K\}$. Here,
parts $V_k \subseteq \mathcal{V}$, for $k = 1, 2, \ldots, K$, are pairwise disjoint and mutually exhaustive. In a partition $\Pi$ of $\mathcal{H}$, a net that connects at least one vertex in a part is said to connect that part. The connectivity set $\Lambda(n)$ of a net $n$ is defined as the set of parts connected by $n$. The connectivity $\lambda(n) = |\Lambda(n)|$ of a net $n$ denotes the number of parts connected by $n$. A net $n$ is said to be cut if it connects more than one part (i.e., $\lambda(n) > 1$), and uncut otherwise (i.e., $\lambda(n) = 1$). The cut and uncut nets are also referred to as external and internal nets, respectively. $\mathcal{N}_{\text{ext}}(V_k)$ denotes the set of external nets of part $V_k$. A vertex is said to be a boundary vertex if it is connected by at least one cut net.

For a $K$-way partition $\Pi$ of a given hypergraph $\mathcal{H}$, imbalance ratio $ibr(\Pi)$ is defined as follows:

$$ibr(\Pi) = \frac{W_{\text{max}}}{W_{\text{avg}}} - 1.$$  

Here, $W_{\text{max}} = \max_{V_k \in \Pi} \{w(V_k)\}$ and $W_{\text{avg}} = W_{\text{tot}}/K$, where $W_{\text{tot}} = w(\mathcal{V})$.

There are various cutsize metrics for representing the cost $\chi(\Pi)$ of a partition $\Pi$. Two most widely used cutsize metrics are given below.

- **Cut-net** metric: The cutsize is equal to the sum of the costs of the cut nets.

  $$\chi(\Pi) = \sum_{n \in \mathcal{N}_{\text{ext}}} c(n)$$  

- **Connectivity** metric: Each cut net $n$ contributes $(\lambda(n) - 1)c(n)$ to the cutsize.

  $$\chi(\Pi) = \sum_{n \in \mathcal{N}_{\text{ext}}} (\lambda(n) - 1)c(n)$$

Given these definitions, the $K$-way hypergraph partitioning problem is defined as follows.

**Definition 1** $K$-Way Hypergraph Partition. *Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, number of parts $K$, a maximum imbalance ratio $\varepsilon$, and a cutsize metric $\chi(\cdot)$; find*
a \( K \)-way partition \( \Pi \) of \( \mathcal{H} \) that minimizes \( \chi(\Pi) \) subject to the balancing constraint \( \text{ibr}(\Pi) \leq \varepsilon \).

This problem is known [46] to be an \( \mathcal{NP} \)-hard problem.

2.2 \( K \)-Way Hypergraph Partitioning With Vertex Replication

For a given \( K \)-way partition \( \Pi \) of \( \mathcal{H} \), \( \mathcal{R}(\Pi) = \{ R_1, R_2, \ldots, R_K \} \) denotes the replication set, where \( R_k \subseteq V \) and \( R_k \cap V_k \neq \emptyset \), for \( k = 1, 2, \ldots, K \). That is, \( R_k \) denotes the subset of vertices added to part \( V_k \) of \( \Pi \) as replicated vertices. Note that replication subsets are possibly pairwise overlapping since a vertex might be replicated in more than one part. The replication set \( \mathcal{R}(\Pi) \) for a given partition \( \Pi \) of \( \mathcal{H} \) induces the following \( K \)-way hypergraph partition with vertex replication:

\[
\Pi_r(\Pi, \mathcal{R}) = \{ V_1^r = V_1 \cup R_1, V_2^r = V_2 \cup R_2, \ldots, V_K^r = V_K \cup R_K \}.
\]

Note that although \( V_k \)'s of \( \Pi \) are pairwise disjoint, \( V_k^r \)'s of \( \Pi_r \) are overlapping. Previously defined \( \chi(\cdot) \) and \( \text{ibr}(\cdot) \) functions are directly applicable to \( \Pi_r \) without any changes. The total weight after replication is defined as \( W_{\text{tot}}^r = W_{\text{tot}} + \sum_{R_k \in \mathcal{R}} w(R_k) \). The main problem addressed in this paper is the following.

**Problem 1** Balance Preserving Min-Cut Replication Set (BPMCRS) for a \( K \)-Way Hypergraph Partition. Given a hypergraph \( \mathcal{H} = (V, \mathcal{N}) \), a \( K \)-way partition \( \Pi \) of \( \mathcal{H} \), and a replication capacity ratio \( \rho \); find a \( K \)-way replication set \( \mathcal{R}(\Pi) \) that minimizes the cutsize \( \chi(\Pi_r) \) of the induced replicated partition \( \Pi_r \) subject to the replication capacity constraint of \( W_{\text{tot}}^r \leq (1 + \rho)W_{\text{tot}} \) and the balancing constraint of \( \text{ibr}(\Pi_r) \leq \text{ibr}(\Pi) \).

Even without the balancing constraint, the min-cut replication set (MCRS) problem is known [38] to be \( \mathcal{NP} \)-hard. Alternative to the proof of Hwang [38],
a simple transformation of the set-union knapsack (SUK) problem – which is known [42] to be $\mathcal{NP}$-hard – to the MCRS problem is presented in Appendix A.

### 2.3 The Dulmage-Mendelsohn Decomposition

The Dulmage-Mendelsohn (DM) decomposition is a canonical decomposition on bipartite graphs and described in a series of papers [28, 29, 30, 39] by Dulmage, Johnson, and Mendelsohn. Pothen and Fan [55] formalized this decomposition by a series of lemmas and explained their enhancements.

A bipartite graph $\mathcal{G} = (\mathcal{V} = R \cup C, \mathcal{E})$ is a graph whose vertex set $\mathcal{V}$ is partitioned into two parts $R$ and $C$ such that the edges in $\mathcal{E}$ connect vertices in two different parts. A matching on a bipartite graph is a subset of its edges without any common vertices. A maximum matching is a matching that contains the largest possible number of edges.

**Definition 2** The Dulmage-Mendelsohn Decomposition. Let $\mathcal{M}$ be a maximum matching for a bipartite graph $\mathcal{G} = (\mathcal{V} = R \cup C, \mathcal{E})$. The Dulmage-Mendelsohn decomposition canonically decomposes $\mathcal{G}$ into three parts

$$\Pi = \{V_H = R_H \cup C_H, V_S = R_S \cup C_S, V_V = R_V \cup C_V\},$$

where $R_H, R_S, R_V$ and $C_H, C_S, C_V$ respectively are subsets of $R$ and $C$ sets with the following definitions based on $\mathcal{M}$:

- $R_V = \{v_i \in R \mid v_i \text{ is reachable by an alternating path from some unmatched vertex } v_j \in R\}$
- $R_H = \{v_i \in R \mid v_i \text{ is reachable by an alternating path from some unmatched vertex } v_j \in C\}$
- $R_S = R - (R_V \cup R_H)$
- $C_V = \{v_i \in C \mid v_i \text{ is reachable by an alternating path from some unmatched vertex } v_j \in R\}$
- $C_H = \{v_i \in C \mid v_i \text{ is reachable by an alternating path from some unmatched vertex } v_j \in C\}$
- $C_S = C - (C_V \cup C_H)$
Following properties given in [54, 55] regarding the $R_H$, $R_S$, $R_V$ and $C_H$, $C_S$, $R_S$ subsets provide certain features related with the structure of the Dulmage-Mendelsohn decomposition. The sets $R_V$, $R_S$, and $R_H$ are pairwise disjoint; similarly, the sets $C_V$, $C_S$, and $C_H$ are pairwise disjoint. A matching edge of $M$ connects: a vertex in $R_V$ only to a vertex in $C_V$; a vertex in $R_S$ only to a vertex in $C_S$; and a vertex in $R_H$ only to a vertex in $C_H$. Vertices in $R_S$ are perfectly matched to vertices in $C_S$. No edge connects: a vertex in $C_H$ to vertices in $R_S$ or $R_V$; a vertex in $C_S$ to vertices in $R_V$. $C_H$ and $R_V$ are the unique smallest sets that maximize the $|C_H| - |R_H|$ and $|R_V| - |C_V|$ differences, respectively. The subsets $R_H$, $R_S$, $R_V$ and $C_H$, $C_S$, $C_V$ are independent of the choice of the maximum matching $M$; hence the Dulmage-Mendelsohn decomposition is a canonical decomposition of the bipartite graph.

For larger bipartite graphs, one might opt for a more fine-grained decomposition. For this purpose, Pothen and Fan [55] further decomposes $R_H$, $R_S$, $R_V$ and $C_H$, $C_S$, $C_V$ sets into smaller subsets. For the simplicity of the forthcoming discussions, the Dulmage-Mendelsohn decomposition will be referred to as coarse-grained decomposition and enhancements of Pothen and Fan will be referred to as fine-grained decomposition.

$G_X$ denotes a bipartite subgraph of $G$, where $X$ is one of $H$, $S$, or $V$. That is, for a given bipartite subgraph $G_X = (V_X = R_X \cup C_X, E_X)$, $E_X$ corresponds to the subset of edges in $E$, which connects either a vertex from $R_X$ to a vertex in $C_X$, or a vertex from $C_X$ to a vertex in $R_X$. The fine-grained decomposition is formalized as follows.

**Definition 3** Fine-Grained Dulmage-Mendelsohn Decomposition. Let $M$ be a maximum matching for a bipartite graph $G = (V = R \cup C, E)$ and $G_H$, $G_S$, $G_V$ be bipartite subgraphs induced by the coarse-grained decomposition of $R$ and $C$ sets into $R_H$, $R_S$, $R_V$ and $C_H$, $C_S$, $C_V$ subsets. Fine-grained decomposition of bipartite subgraphs $G_H$, $G_S$, and $G_V$ is constructed as follows.

- Find connected components in $G_H$ and $G_V$ subgraphs.
- Using $G_S$, construct a new directed bipartite graph $G'_S$, where matched edges
are left undirected, and other unmatched edges are directed from $C_S$ to $R_S$.

Find strongly connected components in $G'_S$.

Depending on the structure of the given bipartite graph and maximum matching, resultant fine-grained decomposition is expected to provide much more number of partitions than its coarse-grained equivalent.

For a given bipartite graph $G = (\mathcal{V}, \mathcal{E})$, a maximum matching can be found in $O(|\mathcal{E}|\sqrt{|\mathcal{V}|})$ time due to Hopcroft-Karp algorithm. In the coarse-grained decomposition phase, a depth-first search is performed for every unmatched vertex for finding alternating paths. Thus, coarse-grained decomposition runs in $O(|\mathcal{E}|\sqrt{|\mathcal{V}|} + O(|\mathcal{V}|(|\mathcal{V}| + |\mathcal{E}|))$ time, that is, in $O(|\mathcal{V}|(|\mathcal{V}| + |\mathcal{E}|))$ time. In the fine-grained decomposition phase, connected components for $G_H$ and $G_V$ can be found in $O(|\mathcal{V}| + |\mathcal{E}|)$ time via breadth-first search and strongly-connected components in $G'_S$ can be found in $O(|\mathcal{V}|(|\mathcal{V}| + |\mathcal{E}|))$ time via Tarjan’s algorithm [64]. Hence, decomposition phase takes $O(|\mathcal{V}|(|\mathcal{V}| + |\mathcal{E}|))$ time in total.
In Fig. 2.1, application of coarse-grained and fine-grained Dulmage-Mendelsohn decompositions are demonstrated on a sample bipartite graph $G = (\mathcal{V} = R \cup C, \mathcal{E})$. This sample hypergraph is composed of 19 vertices and 17 undirected edges. Fig. 2.1b demonstrates a coarse-grained Dulmage-Mendelsohn decomposition of $G$ for a given maximum matching $\mathcal{M}$. Here, matched edges are drawn in black and $V_H$, $V_S$, and $V_V$ parts produced by the coarse-grained decomposition are separated via borders. For instance, $v_3$ is matched with $v_{12}$ and $R_H = \{v_3, v_4\}$ and $C_H = \{v_{11}, v_{12}, v_{13}, v_{14}, v_{15}\}$.

Fig. 2.1c demonstrates a fine-grained decomposition of the sample bipartite graph $G$ in Fig. 2.1a. Here, components are separated via dashed lines. That is, vertices $v_3, v_{11}, v_{12}$ and edges between them constitute a connected component in $G_H$. As seen in Fig. 2.1c, unmatched edges $(v_5, v_{17})$, $(v_6, v_{16})$, and $(v_9, v_{17})$ in $G_S$ are directed from $C_S$ to $R_S$ to construct $G'_S$. There appears two strongly-connected components in $G'_S$: $v_5, v_6, v_{16}, v_{17}$ and $v_9, v_{18}$. 
Chapter 3

Balance Preserving Min-Cut Replication Set

In this chapter, we propose an efficient and effective approach for solving the BPMCRS problem. It is clear that, given a $K$-way partition $\Pi$ of $H$, only the boundary vertices in $\Pi$ have the potential of decreasing the cutsize via replication. Thus, only the boundary vertices are considered for finding a good replication set $R$. In order to be able to handle the balancing constraints on the weights of the parts of the replicated partition, we propose a part-oriented approach by investigating the replications to be performed on each part (in some particular order).

Consider a replication set $R_k$ for a part $V_k$ of $\Pi$. Note that $R_k$ has to maximize the reduction in the cutsize without violating the maximum weight constraint of part $V_k$. It is also clear that, replication of vertices of $R_k$ into part $V_k$ can only decrease the cutsize due to the external nets of part $V_k$. So, while searching for a good $R_k$, we consider only the external nets of part $V_k$ and the boundary vertices of other parts that are connected by the external nets of part $V_k$. That is, we only consider the net set $N_{ext}(V_k)$ and the vertex set $Adj(V_k)$ for finding an $R_k$.

Algorithm 1 displays a general framework for our approach. As seen in the algorithm, for each part $V_k$, we first compute the replication capacity $\kappa_k$ so that
Algorithm 1 FIND_REPLICATION_SET(\(\mathcal{H}, \Pi, W, \rho\))

1: \(\Pi_0^r \leftarrow \Pi\)
2: for \(k \leftarrow 1\) to \(K\) do
3: \(\kappa_k = (1 + \rho)W_{avg} - w(V_k)\)
4: \(\mathcal{H}_k \leftarrow \text{CONSTRUCT}(\mathcal{H}, k, \Pi_{k-1}^r)\)
5: \(\mathcal{H}_k^\text{coarse} \leftarrow \text{COARSEN}(\mathcal{H}_k)\)
6: \(R_k \leftarrow \text{SELECT}(\mathcal{H}_k^\text{coarse}, \kappa_k)\)
7: \(\Pi_k^r \leftarrow \{V_1 \cup R_1, \ldots, V_k \cup R_k, V_{k+1}, \ldots, V_K\}\)
8: \text{UPDATE}(k)\)
9: end for
10: \(\Pi_r \leftarrow \Pi_K^r\)

the initial imbalance will be preserved or improved after the replication. Then, we construct the hypergraph \(\mathcal{H}_k\), which is referred to here as the boundary adjacency hypergraph. Vertices of \(\mathcal{H}_k\) correspond to \(\text{Adj}(V_k)\) and nets of \(\mathcal{H}_k\) are derived from \(N_{\text{ext}}(V_k)\). This hypergraph construction process is described in Section 3.1. After constructing \(\mathcal{H}_k\), a good \(R_k\) is selected from the vertices of \(\text{Adj}(V_k)\) via using an ILP approach described in Section 3.2. In order to reduce the high computation cost of ILP for large \(\mathcal{H}_k\), a novel Dulmage-Mendelsohn decomposition-based coarsening scheme for \(\mathcal{H}_k\) is described in Section 3.3.

### 3.1 Boundary Adjacency Hypergraph Construction

Without loss of generality, here we describe the boundary adjacency hypergraph construction operation to be performed in the \(k\)th iteration of our algorithm for the purpose of deciding on the vertices to be replicated into part \(V_k\). Note that prior to this construction process, the effects of the replications performed in the previous iterations are reflected on \(\Pi_{k-1}^r\) (line 7 of Algorithm 1) and the boundary vertices and cut nets are updated accordingly (line 8 of Algorithm 1). For the simplicity of the forthcoming discussions, we use \(\text{Adj}(V_k)\) and \(N_{\text{ext}}(V_k)\) to refer to the updated adjacency vertex and external net sets of part \(V_k\), respectively. For example, consider an external net \(n_j\) of part \(V_k\) in the original partition \(\Pi_0^r\).
During an earlier iteration \( \ell < k \), if all pins of net \( n_j \) that lie in part \( V_k \) are replicated into part \( V_\ell \), then net \( n_j \) disappears in \( N_{ext}(V_k) \). In such a case, those pins of net \( n_j \) that lie in part \( V_\ell \) and that are only connected by net \( n_j \) to part \( V_k \) disappear from \( Adj(V_k) \).  

**Algorithm 2** \text{\textsc{update}}(k)  
1: \text{for } l \leftarrow (k + 1) \text{ to } K \text{ do}  
2: \text{for each net } n_j \in N_{ext}(V_k) \text{ do}  
3: \text{if } (\text{Pins}(n_j) \cap V_\ell) \cap R_k \neq \emptyset \text{ then}  
4: \text{for each vertex } v \in (\text{Pins}(n_j) \cap V_k) \text{ do}  
5: \text{if } \text{Nets}(v) \cap N_{ext}(V_\ell) = \{n_j\} \text{ then}  
6: Adj(V_\ell) = Adj(V_\ell) - \{v\}  
7: \text{end if}  
8: \text{end for}  
9: N_{ext}(V_\ell) = N_{ext}(V_\ell) - \{n_j\}  
10: \Lambda(n_j) = \Lambda(n_j) - V_\ell \{\text{optional for cut-net metric}\}  
11: \text{end if}  
12: \text{end for}  
13: \text{end for}  

Two distinct boundary adjacency hypergraphs are required to encapsulate the cut-net (Eq. 2.1) and connectivity (Eq. 2.2) cutsize metrics, which will be referred to as \( \mathcal{H}^\text{cut}_k \) and \( \mathcal{H}^\text{con}_k \), respectively. The construction process for the former and latter are depicted in Algorithms 3 and 4, respectively. In both hypergraphs, the vertex set is composed of \( Adj(V_k) \). In both of these hypergraphs, the objective is to find a set of vertices \( R_k \subseteq Adj(V_k) \) to be replicated into part \( V_k \), such that the total cost of nets covered by \( R_k \) is maximized without violating the balance constraint imposed on \( V_k \). The net set definition for \( \mathcal{H}^\text{cut}_k \) and \( \mathcal{H}^\text{con}_k \) should be done in according to this coverage objective. Note that a net \( n_j \) of \( \mathcal{H}^\text{cut}_k / \mathcal{H}^\text{con}_k \) is said to be covered by \( R_k \) if all pins of \( n_j \) in \( Adj(V_k) \) lie within \( R_k \).  

**Algorithm 3** \text{\textsc{construct}}(\mathcal{H}, k, \Pi_{k-1}) \text{ for cut-net metric}  
1: \mathcal{V}^\text{cut}_k \leftarrow Adj(V_k)  
2: \mathcal{N}^\text{cut}_k \leftarrow N_{ext}(V_k)  
3: \text{for each net } n_j \in N_{ext}(V_k) \text{ do}  
4: Pins(n_j) \leftarrow Pins(n_j) - V_k  
5: \text{end for}  
6: \text{return } \mathcal{H}^\text{cut}_k \leftarrow (\mathcal{V}^\text{cut}_k, \mathcal{N}^\text{cut}_k)
For the cut-net metric, in order to reduce the cutsize related with a net $n_j$ in $N_{ext}(V_K)$, the net $n_j$ should be made internal to part $V_k$, which is feasible only when all pins of net $n_j$ in $Adj(V_K)$ are replicated into $V_k$. Thus, the net set of $H^\text{cut}_k$ is selected as the external net set of part $V_k$ (line 2 of Algorithm 3). Since $H^\text{cut}_k$ is used to find the set of vertices to be replicated into part $V_k$, the boundary vertices of part $V_k$ should be extracted from the pin list of the nets of $H^\text{cut}_k$ (lines 3–4 of Algorithm 3).

Algorithm 4 \textsc{construct}(H, k, \Pi_{k-1}^{r}, \kappa_k) for connectivity metric

1: $\mathcal{V}^\text{con}_k \leftarrow Adj(V_k)$
2: $\mathcal{N}^\text{con}_k \leftarrow \emptyset$
3: \textbf{for} each net $n_j \in N_{ext}(V_k)$ \textbf{do}
4: \hspace{1em} \textbf{for} each part $V_\ell \in \Lambda(n_j)$ and $V_\ell \neq V_k$ \textbf{do}
5: \hspace{2em} $\mathcal{N}^\text{con}_k \leftarrow \mathcal{N}^\text{con}_k \cup \{n_\ell^j\}$
6: \hspace{2em} $\text{Pins}(n_\ell^j) \leftarrow \text{Pins}(n_j) \cap V_\ell$
7: \hspace{1em} \textbf{end for}
8: \textbf{end for}
9: \textbf{return} $H^\text{con}_k \leftarrow (\mathcal{V}^\text{con}_k, \mathcal{V}^\text{con}_k)$

For the connectivity metric, in order to reduce the cutsize related with a net $n_j$ in $N_{ext}(V_K)$, it is sufficient to replicate a subset of the pins of net $n_j$ so that $\lambda(n_j)$ in $\Pi'$ will decrease. That is, number of parts connected by net $n_j$ will decrease after the replication. For this reason, the nets of $H^\text{con}_k$ is derived from $N_{ext}(V_k)$ by applying a net splitting operation to each external net in such a way that each external net $n_j$ is split into $\lambda(n_j) - 1$ new nets. This splitting operation is performed as follows: For each net $n_j$ in $N_{ext}(V_k)$, we traverse over the connectivity set $\Lambda(n_j)$ of $n_j$ and introduce a new net $n_\ell^j$ for each part $V_\ell \neq V_k$ in $\Lambda(n_j)$. The newly introduced net $n_\ell^j$ is set to connect only those pins of $n_j$ that lie in part $V_\ell$ (lines 4–6 of Algorithm 4).

Fig. 3.1 shows a 3-way partition of a sample hypergraph $H$ with 24 boundary vertices and 19 cut nets. In figures, circles denote vertices and dots denote nets, where a number $i$ in a circle denotes a vertex $v_i$ and a number $j$ besides a dot denotes a net $n_j$. Note that only boundary vertices and cut nets are numbered for the sake of simplicity. Fig. 3.2 shows the boundary adjacency hypergraphs $H^\text{cut}_1$ (Fig. 3.2a) and $H^\text{con}_1$ (Fig. 3.2b) for part $V_1$ for cut-net and connectivity metrics,
Figure 3.1: A 3-way partition of a sample hypergraph $\mathcal{H}$.

(a) Boundary adjacency hypergraph $\mathcal{H}_{1}^{\text{cut}}$ of part $V_1$.

(b) Boundary adjacency hypergraph $\mathcal{H}_{1}^{\text{con}}$ of part $V_1$.

Figure 3.2: Sample boundary adjacency hypergraph construction.
respectively. Comparing Fig. 3.1, with Figs. 3.2a and 3.2b shows that $V_2$’s and $V_3$’s boundary vertices $v_5, v_6, \ldots, v_{19}$ that are connected by at least one external net of $V_1$ constitute the vertices of both $H_1^{cut}$ and $H_1^{con}$.

Comparing Fig. 3.1 with Figs. 3.2a and 3.2b shows that each of the external nets $n_1, n_2, \ldots, n_{13}$ of $V_1$ incurs a net in $H_1^{cut}$. Similarly, each of the external nets $n_1, n_2, \ldots, n_6$ and $n_{11}, n_{12}, n_{13}$ of $V_1$, which have a connectivity of 2, incurs a single net in $H_1^{con}$. On the other hand, each of the external nets $n_7, n_8, n_9, n_{10}$ of $V_1$, which have a connectivity of 3, incurs 2 nets in $H_1^{con}$. For example, $n_7$ with $Pins(n_7) = \{v_{10}, v_{14}, v_{16}\}$ connects both parts $V_2$ and $V_3$, and it incurs two nets $n_7^2$ and $n_7^3$ in $H_1^{con}$, where $Pins(n_7^2) = \{v_{10}\}$ and $Pins(n_7^3) = \{v_{14}, v_{16}\}$. Note that $n_7^2$ and $n_7^3$ are respectively shown as $7^2$ and $7^3$ in Fig. 3.2b.

As seen in Fig. 3.2a, net $n_9$ of $H_1^{cut}$ is covered by the vertex set $\{v_9, v_{10}, v_{11}\}$. So, the cut-net cutsize related with net $n_9$ can be reduced only if all of the vertices $v_9, v_{10}, v_{11}$ are replicated into part $V_1$. On the other hand, as seen in Fig. 3.2b, net $n_9^2$ of $H_1^{con}$ is covered by the vertex set $\{v_9, v_{10}\}$ and $n_9^3$ of $H_1^{con}$ is covered by the vertex set $\{v_{11}\}$. So, the connectivity cutsize related with net $n_9$ can be reduced by 1 (assuming unit net costs) either by replicating vertices $v_9$ and $v_{10}$ into part $V_1$ or by replicating vertex $v_{11}$ into part $V_1$. Note that, although the vertex set $\{v_9, v_{10}, v_{11}\}$ covers only net $n_9$ in $H_1^{cut}$, it covers nets $n_9^2, n_7^3, n_8^3,$ and $n_9^3$ in $H_1^{con}$. So, replicating the vertex set $\{v_9, v_{10}, v_{11}\}$ into part $V_1$ reduces the cut-net cutsize by 1, whereas, it reduces the connectivity cutsize by 4.
In the first iteration of Algorithm 1, each net splitting is unique in $\mathcal{H}_{1}^{con}$, since there are no replicated vertices. However, in the following iterations of Algorithm 1, net splittings may not be unique for the further $\mathcal{H}_{k}^{con}$ constructions because of the replicated vertices. That is, multiple copies of a vertex induces multiple pin selection options for a net. And each different pin selection induces a different net splitting in the boundary adjacency hypergraph. Fig. 3.3 shows this pin selection problem that occurs in the construction of $\mathcal{H}_{k}^{con}$, where vertex $v_4$ was replicated into part $V_m$ in the $m$th iteration for $m < k$. Figs. 3.3b and 3.3c show two possible selections for net $n_1$, which connects to the replicated vertex $v_4$. In Fig. 3.3b, replication of $v_4$ and $v_5$ appear to be necessary to cover $n_1$, whereas, in Fig. 3.3c, replication of $v_5$ is sufficient to cover $n_1$. As depicted in Fig. 3.3, pin selections of nets directly affect the minimization of the number of replicated vertices for covering a particular net. In our proposed model, for a net $n_j$ and vertex $v_j \in Pins(n_j)$, if there exists a copy of $v_j$ in part $V_k$ that was previously replicated into $V_k$ for the purpose of decreasing the connectivity set of $\lambda(n_j)$, then $n_j$ is connected to $v_j$ in part $V_k$; otherwise, it is connected to the $v_j$ in part $V_k$ that is provided by the initial partitioning. For instance, in Fig. 3.3, if $v_4$ is replicated to part $V_m$ in a previous iteration for $n_1$, then $n_1$ is connected to $v_4$ in $R_m$; otherwise, it is connected to $v_4$ in $V_\ell$.

### 3.2 Vertex Selection in Boundary Adjacency Hypergraph

In our approach, boundary adjacency hypergraph $\mathcal{H}_k = (\mathcal{V}_k, \mathcal{N}_k)$ is derived from the cut nets of part $V_k$ and the adjacent vertices to part $V_k$. Since nets in $\mathcal{H}_k^{cut}$ and $\mathcal{H}_k^{con}$ correspond to the cut nets for the cut-net and connectivity cutsize metrics, covering these nets has a direct effect on the cutsize related with part $V_k$. Hence, it is clear that only the vertices in $\mathcal{V}_k$ have the potential of decreasing the cutsize related with part $V_k$ via replication. In this section, our objective is the optimal selection of a subset $R_k$ of vertices in $\mathcal{V}_k$ that are to be replicated into part $V_k$. Optimality in this context is defined as, given boundary adjacency hypergraph
\( \mathcal{H}_k \) and maximum replication capacity \( \kappa_k \), selecting a subset \( R_k \) of vertices in \( V_k \) that maximize the sum of the costs of the covered nets under a given capacity constraint of \( w(R_k) \leq \kappa_k \). This net coverage objective corresponds to the set-union knapsack problem [42] (SUKP). (See Appendix A for details.) We provide an ILP formulation for this problem as follows.

\[
\begin{align*}
\text{maximize} & \quad \sum_{n_j \in N_k} c(n_j) x(n_j) \\
\text{subject to} & \quad |Pins(n_j)| x(n_j) \leq \sum_{v_i \in Pins(n_j)} y(v_i) \quad \text{for } \forall n_j \in \mathcal{N}_k \quad (3.2) \\
& \quad \sum_{v_i \in V_k} w(v_i) y(v_i) \leq \kappa_k \quad (3.3)
\end{align*}
\]

where \( x(n_j) = \begin{cases} 1, & \text{if net } n_j \text{ is covered} \\ 0, & \text{otherwise} \end{cases} \)

\( y(v_i) = \begin{cases} 1, & \text{if vertex } v_i \text{ is selected} \\ 0, & \text{otherwise} \end{cases} \)

Binary variable \( x(n_j) \) is set to 1, if a net \( n_j \) is covered by the selected vertices. Likewise, if a vertex \( v_i \) is selected for replication, binary variable \( y(v_i) \) is set to 1. Objective (3.1) tries to maximize the sum of the cost \( c(n_j) \) of every covered net for which \( x(n_j) = 1 \). Inequality (3.2) constrains a net \( n_j \) to be covered if all of its pins are selected, i.e., net \( n_j \) is covered if \( y(v_i) = 1 \) for every \( v_i \in Pins(n_j) \). In expression (3.3), the sum of the weights of the selected vertices are constrained by \( \kappa_k \). Since there are no restrictions on vertex replications, but inequality (3.3), formulation might produce redundant vertex replications as much as \( \kappa_k \) allows. That is, for certain vertices \( v_i \), \( y(v_i) \) can be set to 1, where \( v_i \) is not contained by the adjacencies of the covered nets. But once the set of \( x(n_j) \) is computed, necessary \( y(v_i) \) values can be extracted from \( Pins(n_j) \) without allowing any redundant vertex replications.

In given ILP formulation, for each boundary adjacency hypergraph \( \mathcal{H}_k \), there are \(|V_k| + |\mathcal{N}_k|\) variables for \( x(n_j) \) and \( y(n_j) \), and \(|\mathcal{N}_k| + 1\) constraints (inequalities (3.2) and (3.3)), and a single maximization objective.
ILP model formalized in expressions (3.1)–(3.3) provides the optimal net coverage for a given boundary adjacency hypergraph $\mathcal{H}_k$ and maximum replication capacity $\kappa_k$. In Appendix A, the relation between set-union knapsack problem and net coverage in boundary adjacency hypergraph is detailed and it is proved that the net coverage problem is an $\mathcal{NP}$-hard problem. Hence, from a practical point of view, this formulation is expected to consume a significant amount of time as the sum of input variables – $|V_k|$ and $|N_k|$ – increase in size. To reduce this high computation cost of the ILP phase, below preprocessing procedures are introduced and applied to $\mathcal{H}_k$ at each iteration before the vertex selection process for replication.

1. Remove infeasible nets (that $\kappa_k$ isn’t sufficient to cover via vertex replication) and the vertices that are only connected by such nets.

2. Use heuristics to coarsen boundary adjacency hypergraph into a smaller hypergraph.

3. Restrict ILP solver running time to a certain duration.

### 3.3 Coarsening of Boundary Adjacency Hypergraph

In order to reduce the high computation cost of the ILP phase, we propose an effective coarsening approach based on the Dulmage-Mendelsohn decomposition. At $k$th iteration of the algorithm, we coarsen the boundary adjacency hypergraph $\mathcal{H}_k$ to $\mathcal{H}_k^{\text{coarse}}$. Then, instead of $\mathcal{H}_k$, we pass this $\mathcal{H}_k^{\text{coarse}}$ to the ILP solver.

The Dulmage-Mendelsohn decomposition operates on bipartite graphs $\mathcal{G} = (V = R \cup C, E)$, hence, each boundary adjacency hypergraph $\mathcal{H}_k = (V_k, N_k)$ is represented in terms of its bipartite graph equivalent $\mathcal{G}_k = (V_k = R_k \cup C_k, E_k)$ for coarsening. Vertices $V_k$ and nets $N_k$ in $\mathcal{H}_k$ constitute the $R_k$ and $C_k$ sets in $\mathcal{G}_k$, respectively. That is, for a vertex $v_i \in V_k$ there is a corresponding vertex $v_{v_i} \in R_k$ and for a net $n_j \in N_k$ there is a corresponding vertex $v_{n_j} \in C_k$. Pins
between nets and vertices constitute the edge set $\mathcal{E}_k$ of $\mathcal{G}_k$. That is, for a net $n_j \in \mathcal{N}_k$ and $v_i \in \text{Pins}(n_j)$ there is an undirected edge $(v_i, v_{n_j})$ in $\mathcal{E}_k$. After the decomposition, clusters in $\mathcal{G}_k$ are easily projected back to $\mathcal{H}_k$ by reversing back the transformation.

Vertex selection in boundary adjacency hypergraph is constrained by the total weight of the selected vertices for replication and its objective is to maximize the cost of the nets covered. Thus, our objective in the coarsening phase is to cluster vertices and nets in such a way that the vertex groups with similar net coverage characteristics get clustered together. Characterization in this context is intuitively estimated as a ratio between the number of vertices in the cluster and the nets covered by these vertices. That is, clusters with small number of vertices covering a large number of nets correspond to the high-quality replications; clusters with average number of vertices covering an average number of nets correspond to the mid-quality replications; and, clusters with large number of vertices covering a small number of nets correspond to the low-quality replications. As described in Section 2.3, the coarse-grained Dulmage-Mendelsohn decomposition states that $C_H$ and $R_V$ are the unique smallest sets that maximize the $|C_H| - |R_H|$ and $|R_V| - |C_V|$ differences and $|R_S| = |C_S|$. We know that every boundary adjacency hypergraph $\mathcal{H}_k$ can be represented as a bipartite graph $\mathcal{G}_k$. Hence, we can use fine-grained Dulmage-Mendelsohn decomposition to encapsulate the replication characteristics of the original hypergraph into its coarsened representation, where components in $R_H$ correspond to high-quality replications, components in $R_S$ correspond to mid-quality replications, and components in $R_V$ correspond to low-quality replications.

In Section 2.3, it is shown that the coarse-grained and fine-grained Dulmage-Mendelsohn decomposition runs in $O(|\mathcal{V}|(|\mathcal{V} | + |\mathcal{E}|))$ time in total. In case of $\mathcal{G}_k = (\mathcal{V}_k = R_k \cup C_k, \mathcal{E}_k)$ bipartite graph representation of the boundary adjacency hypergraph, this value is equal to $O(|\mathcal{V}_k|(|\mathcal{V}_k | + |\mathcal{E}_k|))$. And from the relation between $R_k, C_k$ and $\mathcal{V}_k, \mathcal{N}_k$, it becomes $O((|\mathcal{V}_k| + |\mathcal{N}_k|)(|\mathcal{V}_k | + |\mathcal{N}_k|) + \sum_{n_j \in \mathcal{N}_k} |\text{Pins}(n_j)|)).$
Figure 3.4: The fine-grained Dulmage-Mendelsohn decomposition of sample boundary adjacency hypergraph $\mathcal{H}_1^{\text{con}}$. 
Fig. 3.4a demonstrates a simplified drawing of the boundary adjacency hypergraph $\mathcal{H}^\text{con}_1$ given in Fig. 3.2b. Fig. 3.4b demonstrates the coarse-grained and fine-grained Dulmage-Mendelsohn decomposition of $\mathcal{H}^\text{con}_1$. In Fig. 3.4b, since parts $V_2$ and $V_3$ are disjoint, it is possible to apply the Dulmage-Mendelsohn decomposition separately to parts $V_2$ and $V_3$. Components in Fig. 3.4b constitute the new vertices and nets in Fig. 3.4c. For instance, the 3rd component composed of vertices $v_{14}, v_{15}$ and nets $n^3_7, n^3_8$ in Fig. 3.4b constitute the vertex $v_3$ and net $n_3$ in the coarsened hypergraph in Fig. 3.4c.

### 3.4 Balance Preserving Replication Capacity Computation

Maximum replication capacity $\kappa_k$ represents the amount of replication allowed into part $V_k$. Note that the maximum replication capacity $\kappa_k$ of each part $V_k$ directly affects the contribution of $R_k$ to the partition imbalance. That is, even a single miscalculated $\kappa_k$ might result in a significant change in the imbalance of the whole partition. Hence, maximum replication capacity of each part must be chosen in such a way that, after the replication, imbalance of the partition is preserved and the replication capacity is consumed to reduce the cutsize as much as possible. For this purpose, we set $\kappa_k$ to $(1 + \rho)W_{\text{avg}} - w(V_k)$ for each part $V_k$. That is, we aim to raise the weight of part $V_k$, i.e., $w(V_k)$, to the average weight of a part after all available replication capacity is consumed, i.e., $(1 + \rho)W_{\text{avg}}$. Since replication introduces new vertices to the parts, this scheme will just increase the weight of the parts that are smaller than $(1 + \rho)W_{\text{avg}}$. Hence, partition imbalance changes as follows.

- If $(1 + \rho)W_{\text{avg}} < W_{\text{max}}$, after the replication, $W_{\text{avg}}$ is expected to increase, while $W_{\text{max}}$ stays the same. Hence, balance will stay the same even in the worst case, that is, no replication; otherwise, balance will be improved.

- Otherwise, we have enough room to raise the total weight of each part to $(1 + \rho)W_{\text{avg}}$. That is, even if the replication doesn’t consume all available
capacity and increase the imbalance, we can reduce the final imbalance to its initial value by making dummy vertex replications without considering any net coverages.

At $k$th iteration of the algorithm, we try to raise $w(V_k)$ to $(1 + \rho)W_{avg}$, which corresponds to the part weight of an optimally balanced partition. Hence, after the replication, a significant reduction in the partition imbalance ratio is highly expected. This observation unsurprisingly holds with the experimental results as well.

### 3.5 Part Visit Order for Replication

In our approach, parts are visited in some particular order and replication set $R_k$ of a part $V_k$ directly affects the boundary adjacency graph $H_\ell$ for $\ell > k$. Hence, ordering of the parts plays an important role considering the global quality of the proposed scheme. This effect can be observed both in the cutsize and imbalance reduction. For instance, processing parts in increasing weight order might result in poor imbalance reductions. That is, most of the replication capacity might be consumed by larger parts and $W_{max} - W_{avg}$ difference could not be reduced as expected. Moreover, one would intuitively select parts whose average boundary adjacency hypergraph net degree is smaller compared to others. That is, consider a net $n_j$ connected to $m$ vertices in part $V_k$ and $n$ vertices in part $V_\ell$. If $m < n$, part ordering should be done in such a way that $V_\ell$ should be processed first to cover net $n_j$ with the least possible number of replications, i.e., $m$ vertices. In the experimentation results, comparison of evaluated ordering schemes are given.
Chapter 4

Experimental Results

In this chapter, experimental results evaluated for various data set collections are given. First, in Section 4.1, experimented data set collections are detailed. Next, implementation details are given in Section 4.2. In Section 4.3, we present the results regarding the initial partitions of the data sets. Then, in Section 4.4, the replication results for cutsize and imbalance reductions are given. Next, in Sections 4.5 and 4.6, we discuss the effect of coarsening and part ordering schemes over replication results. Finally, we present system resource usage statistics of an implementation of the proposed model in Section 4.7.

4.1 Data Set Collection

There are various hypergraph models successfully incorporated into spatial database [26, 25] and information retrieval [15] systems. For experimentation purposes, we used sample hypergraphs from these domains and investigated the effect of replication in these hypergraph models.

To investigate the effect of replication in spatial databases, a wide range of real-life road network (RN) data sets are collected from US Tiger/Line [11] (Minnesota7 including 7 counties Anoka, Carver, Dakota, Hennepin, Ramsey, Scott,
CHAPTER 4. EXPERIMENTAL RESULTS

Washington; San Francisco; Oregon; New Mexico; Washington), US Department of Transportation [52] (California Highway Planning Network), and Brinkhoff’s network data generator [10] (Oldenburg; San Joaquin). Hypergraph models of RN data sets are constructed according to the clustering hypergraph model presented in [26].

To examine the effect of replication in information retrieval systems, text crawls are downloaded from the Stanford WebBase project[1, 21] (CalGovernor, Facebook, Wikipedia) and University of Florida Sparse Matrix Collection [24] (Stanford). Stanford data set represents links in a set of crawled web pages. In hypergraph models of Stanford WebBase project data sets, terms correspond to vertices and documents correspond to nets. This construction scheme is detailed in [15].

Properties of the hypergraphs extracted from the collected data sets are presented in Table 4.1. Column explanations of the hypergraph properties table are as follows.

<table>
<thead>
<tr>
<th>Column</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Pins</td>
</tr>
<tr>
<td>$d_{avg}^{\mathcal{N}}$</td>
<td>Average net degree.</td>
</tr>
<tr>
<td>$c_{avg}$</td>
<td>Average net cost.</td>
</tr>
<tr>
<td>$d_{avg}^{\mathcal{V}}$</td>
<td>Average vertex degree.</td>
</tr>
<tr>
<td>$w_{avg}$</td>
<td>Average vertex weight.</td>
</tr>
</tbody>
</table>

In Table 4.1, hypergraphs are grouped by their domains (RN and IR) and sorted in increasing $|Pins|$ order. As seen in the table, RN hypergraphs have relatively small average net degrees. This may give us the intuition that in RN data sets, covering a net is likely to be easier compared to IR data sets. In IR hypergraphs, large $|Pins|$ and $d_{avg}^{\mathcal{N}}$ values show that the boundary adjacency hypergraphs are expected to be quite large in size, hence, it is expected that coarsening will play an important role in these hypergraphs.
CHAPTER 4. EXPERIMENTAL RESULTS

| Type | H | |V| | |N| | |Pins| | |d|N|avg| |c|avg| | |d|V|avg| |w|avg |
|------|---|---|---|---|---|---|---|---|---|---|---|---|
| RN   | Oldenburg | 5389 | 13003 | 32945 | 2.5 | 8.4 | 6.1 | 46.9 |
|      | California | 14185 | 33414 | 94857 | 2.8 | 6.7 | 6.7 | 53.3 |
|      | SanJoaquin | 22987 | 44944 | 131603 | 2.9 | 8.3 | 5.7 | 52.5 |
|      | Minnesota | 46103 | 78371 | 239422 | 3.1 | 13.1 | 5.2 | 53.5 |
|      | SanFrancisco | 213371 | 319305 | 967917 | 3.0 | 9.1 | 4.5 | 51.5 |
|      | Wyoming | 317100 | 512754 | 1443433 | 2.8 | 8.9 | 4.6 | 49.0 |
|      | NewMexico | 556115 | 781219 | 2270120 | 2.9 | 8.9 | 4.1 | 49.5 |
|      | Oregon | 601672 | 811166 | 2332870 | 2.9 | 9.5 | 3.9 | 48.3 |
|      | Washington | 652063 | 824650 | 2427615 | 2.9 | 11.7 | 3.7 | 49.0 |
| IR   | Stanford | 281903 | 281903 | 2312497 | 8.2 | 1.0 | 8.2 | 8.2 |
|      | CalGovernor | 92279 | 30805 | 3004908 | 97.5 | 1.0 | 32.6 | 1.0 |
|      | Facebook | 4618974 | 66568 | 14277456 | 214.5 | 1.0 | 3.1 | 1.0 |
|      | Wikipedia | 1350762 | 70115 | 43285851 | 617.4 | 1.0 | 32.0 | 1.0 |

Table 4.1: Data set properties.

Replication capacity is calculated by $\rho |V| w_{avg}$ and a high capacity will intuitively result in more replications covering more nets. Hence, low values of $|V| w_{avg}$ is expected to produce relatively poor results in replication. For instance, Oldenburg and CalGovernor is highly expected to fall in this area. However, this case is not likely to be applicable for others.

4.2 Implementation Details

Conducted replication experiments are evaluated on a Debian GNU/Linux 5.0.4 (x86_64) system running on an AMD Opteron (2.1 GHz) processor. During tests, ANSI C sources are compiled using gcc bundled with 4.3 release of GNU Compiler Collections where -O3 -pipe -fomit-frame-pointer flags are turned on. IBM ILOG CPLEX 12.1 is used in single-threaded mode to solve ILP problems. ILP pass for each boundary adjacency hypergraph is time limited to 200 milliseconds. PaToH [20] v3.1 is used with default parameters for initial partitioning of the data sets. Coarsening is disabled for boundary adjacency hypergraphs where total number of pins are smaller than or equal to 30.
CHAPTER 4. EXPERIMENTAL RESULTS

4.3 Initial $K$-Way Hypergraph Partitioning

In the BPMCRS problem, it is assumed that an initial partition of the supplied hypergraph is provided. For this purpose, we partitioned the hypergraphs for two different $K$ values – 128 and 256 – via PaToH. In Table 4.2, partition properties of the test hypergraphs are given. In this table, columns correspond to the particular properties of partitions as follows.

<table>
<thead>
<tr>
<th>Column</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\chi(\Pi)$</td>
<td>Connectivity cutsize.</td>
</tr>
<tr>
<td>$ibr(\Pi)$</td>
<td>Imbalance ratio.</td>
</tr>
<tr>
<td>$</td>
<td>N^*</td>
</tr>
<tr>
<td>$d_{avg}^N$</td>
<td>Average cut net degree.</td>
</tr>
<tr>
<td>$c_{avg}$</td>
<td>Average cut net cost.</td>
</tr>
<tr>
<td>$\lambda_{avg}$</td>
<td>Average cut net connectivity.</td>
</tr>
<tr>
<td>$</td>
<td>V^*</td>
</tr>
<tr>
<td>$d_{avg}^V$</td>
<td>Average boundary vertex degree.</td>
</tr>
<tr>
<td>$w_{avg}$</td>
<td>Average boundary vertex weight.</td>
</tr>
</tbody>
</table>

For RN data sets, where $d_{avg}^N$ is approximately the same, the correlation between $|Pins|$ in Table 4.1 and $\chi(\Pi)$ in Table 4.2 points out that the connectivity cutsize increases proportional to the total number of pins. However, for IR data sets, varying $d_{avg}^N$ values also affect the $\chi(\Pi)$ and we observe that high $d_{avg}^N$ values generally imply high $\chi(\Pi)$ values.

4.4 Replication Results

In Table 4.3, replication results are listed for hypergraph partitions given in Table 4.2. Column explanations of the Table 4.3 are as follows.
CHAPTER 4. EXPERIMENTAL RESULTS

| Type  | $K$ | $\mathcal{H}$     | $\chi(\Pi)$ | $ibr(\Pi)$ | $|N^*|$ | $d_{\text{avg}}^N$ | $e_{\text{avg}}^*$ | $|\mathcal{V}^*|$ | $d_{\text{avg}}^\mathcal{V}$ | $w_{\text{avg}}^*$ |
|-------|-----|-------------------|-------------|------------|-------|-----------------|-------------------|----------------|-----------------------------|-----------------|
|       | 128 | Oldenburg         | 15377       | 4.3        | 1993  | 2.8             | 7.5               | 1498           | 7.1                          | 50.3            |
|       |     | California        | 21404       | 5.3        | 3661  | 3.2             | 5.7               | 3016           | 7.3                          | 56.1            |
|       |     | SanJoaquin        | 27939       | 25.4       | 3709  | 3.2             | 7.4               | 3136           | 7.2                          | 57.7            |
|       |     | Minnesota         | 40108       | 97.4       | 3719  | 3.3             | 10.7              | 3408           | 6.8                          | 59.1            |
|       |     | SanFrancisco      | 44986       | 5.6        | 6255  | 3.3             | 7.2               | 6194           | 6.2                          | 56.3            |
|       |     | Wyoming           | 46421       | 5.0        | 6527  | 3.0             | 7.1               | 6599           | 5.6                          | 51.0            |
|       |     | NewMexico         | 44386       | 4.2        | 6505  | 3.1             | 6.8               | 6896           | 5.4                          | 51.9            |
|       |     | Oregon            | 51154       | 5.0        | 7079  | 3.0             | 7.2               | 7463           | 5.3                          | 51.3            |
|       |     | Washington        | 58721       | 12.0       | 6621  | 3.1             | 8.8               | 7059           | 5.4                          | 53.1            |
|       | 256 | Oldenburg         | 24148       | 5.3        | 3068  | 2.8             | 7.6               | 2224           | 7.1                          | 50.2            |
|       |     | California        | 34254       | 5.5        | 5535  | 3.3             | 5.9               | 4554           | 7.2                          | 56.1            |
|       |     | SanJoaquin        | 44961       | 13.4       | 5777  | 3.2             | 7.6               | 4871           | 7.1                          | 57.8            |
|       |     | Minnesota         | 66581       | 177.1      | 6088  | 3.3             | 10.8              | 5595           | 6.8                          | 59.0            |
|       |     | SanFrancisco      | 72541       | 3.9        | 9972  | 3.3             | 7.2               | 10021          | 6.2                          | 56.8            |
|       |     | Wyoming           | 69708       | 32.3       | 9829  | 3.0             | 7.1               | 9935           | 5.6                          | 51.5            |
|       |     | NewMexico         | 73516       | 4.3        | 10489 | 3.1             | 7.0               | 11181          | 5.4                          | 52.6            |
|       |     | Oregon            | 77227       | 4.2        | 10678 | 3.1             | 7.2               | 11386          | 5.4                          | 52.1            |
|       |     | Washington        | 91527       | 5.4        | 10132 | 3.2             | 9.0               | 10994          | 5.4                          | 53.8            |
|       | 128 | Stanford          | 15904       | 228.2      | 9181  | 116.2           | 1.0               | 167826         | 11.0                         | 11.0            |
|       |     | CalGovernor       | 201391      | 5.7        | 24476 | 119.6           | 1.0               | 92275          | 32.6                         | 1.0             |
|       |     | Facebook          | 324393      | 1.4        | 58467 | 234.7           | 1.0               | 4611479        | 3.1                          | 1.0             |
|       |     | Wikipedia         | 1040098     | 4.2        | 69117 | 623.5           | 1.0               | 1350568        | 32.0                         | 1.0             |
|       | 256 | Stanford          | 24408       | 777.3      | 12523 | 93.2            | 1.0               | 173321         | 10.9                         | 10.9            |
|       |     | CalGovernor       | 298223      | 5.1        | 27724 | 107.4           | 1.0               | 92278          | 32.6                         | 1.0             |
|       |     | Facebook          | 415405      | 1.2        | 61934 | 225.7           | 1.0               | 4617453        | 3.1                          | 1.0             |
|       |     | Wikipedia         | 1470241     | 4.9        | 69608 | 620.5           | 1.0               | 1350736        | 32.0                         | 1.0             |

Table 4.2: Properties of hypergraph partitions.
As seen in Table 4.3, since $d_{avg}^{N^*}$ values are approximately the same for RN data sets, in a majority of the tests $|\mathcal{V}|$ variable dominates the effect on the quality of the replication. That is, compared to other RN hypergraphs, low $|\mathcal{V}|$ values of Oldenburg hypergraph resulted in low quality replications due to the low replication capacity of $\rho|\mathcal{V}|w_{avg}$. On the other hand, for RN hypergraphs with high $|\mathcal{V}|$ values – i.e., Wyoming, NewMexico, Oregon, and Washington – replication removed almost every net from the cut. For 128-way RN hypergraph partitions, a replication amount of 1%, provides 51.8% reduction in the connectivity cutsize and 16.1% reduction in the imbalance ratio on the average. Same amount of replication provides 56.7% reduction in the connectivity cutsize and 16.2% reduction in the imbalance ratio for 256-way hypergraph partitions. By looking at these improvements, it can be concluded that RN hypergraphs are quite suitable for replication.

For IR data sets, since $d_{avg}^{N^*}$ values of the IR hypergraph partitions are much larger than those of the RN hypergraphs and high replication percentages are common practice in IR systems, replication is evaluated with higher values – 10% and 20% – of $\rho$. Compared to RN hypergraph partitions, both $|\mathcal{V}|$ and $d_{avg}^{N^*}$ values are quite varying among IR hypergraph partitions and both have a more prominent effect on the quality of the replication. For instance, the effect of high $|\mathcal{V}|$ and low $d_{avg}^{N^*}$ values of Facebook is distinctive in the replication results. To conclude, replication can yield promising results depending on the structure of the hypergraph, which can be estimated by simple observations in $|\mathcal{V}|$ and $d_{avg}^{N^*}$.
## Chapter 4. Experimental Results

Table 4.3: Replication results.

| Type | $\rho$ | $K$ | $H$  | $\chi(\%)$ | $ibr(\%)$ | $|Pins(H_k)|$ | $|Pins(\%)|$ |
|------|-------|-----|------|-----------|-----------|----------------|----------------|
| RN   | 0.01  | 128 | Oldenburg | 9.7 | 24.0 | 16.1 | 13.0 |
|      |       |     | California | 15.4 | 19.8 | 54.8 | 70.6 |
|      |       |     | SanJoaquin | 18.3 | 4.9  | 50.8 | 70.5 |
|      |       |     | Minnesota  | 37.3 | 2.0  | 73.1 | 72.9 |
|      |       |     | SanFrancisco | 73.8 | 18.6 | 94.2 | 48.2 |
|      |       |     | Wyoming     | 99.0 | 20.8 | 84.3 | 30.5 |
|      |       |     | NewMexico   | 99.7 | 24.7 | 80.7 | 26.5 |
|      |       |     | Oregon      | 100.0| 20.7 | 89.7 | 24.0 |
|      |       |     | Washington  | 100.0| 9.3  | 86.5 | 14.3 |
|      | 256   |     | Oldenburg   | 6.1  | 19.6 | 12.1 | 0.0  |
|      |       |     | California  | 9.6  | 18.9 | 22.7 | 31.3 |
|      |       |     | SanJoaquin  | 13.7 | 8.4  | 36.4 | 62.4 |
|      |       |     | Minnesota   | 22.4 | 1.5  | 51.5 | 71.7 |
|      |       |     | SanFrancisco| 45.4 | 26.2 | 88.3 | 72.7 |
|      |       |     | Wyoming     | 71.5 | 4.1  | 67.2 | 49.1 |
|      |       |     | NewMexico   | 98.5 | 23.9 | 71.7 | 41.4 |
|      |       |     | Oregon      | 99.9 | 24.5 | 70.7 | 39.5 |
|      |       |     | Washington  | 99.2 | 19.2 | 70.0 | 34.6 |
| IR   | 0.10  | 128 | Stanford    | 49.6 | 13.1 | 2101.1| 368.0 |
|      |       |     | CalGovernor | 3.8  | 100.0| 7989.2| 93.5 |
|      |       |     | Facebook    | 44.6 | 100.0| 649449.1| 98.7 |
|      |       |     | Wikipedia   | 11.4 | 100.0| 3159094.2| 98.6 |
|      | 256   |     | Stanford    | 69.0 | 24.0 | 3348.7| 422.7 |
|      |       |     | CalGovernor | 1.7  | 100.0| 2082.7| 90.9 |
|      |       |     | Facebook    | 45.6 | 100.0| 415554.9| 98.3 |
|      |       |     | Wikipedia   | 7.6  | 100.0| 557740.3| 97.8 |
| IR   | 0.20  | 128 | Stanford    | 39.0 | 10.3 | 1111.0| 182.5 |
|      |       |     | CalGovernor | 9.7  | 100.0| 36296.2| 96.2 |
|      |       |     | Facebook    | 57.8 | 100.0| 612529.1| 98.7 |
|      |       |     | Wikipedia   | 18.2 | 100.0| 5732360.4| 99.0 |
|      | 256   |     | Stanford    | 52.9 | 18.8 | 1638.6| 267.6 |
|      |       |     | CalGovernor | 5.1  | 100.0| 6596.4 | 92.8 |
|      |       |     | Facebook    | 59.2 | 100.0| 383360.9| 98.4 |
|      |       |     | Wikipedia   | 15.7 | 100.0| 2217957.5| 98.0 |
values.

In Table 4.3, $ibr(\%)$ column gives the reduction in the imbalance ratio of the partition in percentages. For RN data sets, small $\rho|V|w_{avg}$ doesn’t provide enough replication capacity to improve the balance. Average partition imbalance reduction is around 16% for RN hypergraphs. Since IR data sets provide considerable amounts of replication capacity, average partition imbalance reduction is around 100% for IR data sets. To summarize, replication provides significant imbalance reductions in a majority of the conducted experiments.

In Table 4.3, the last two columns provide information about the average size of the constructed boundary adjacency hypergraphs and the effect of the coarsening on these hypergraphs. For RN data sets, coarsening reduced the size of the constructed boundary adjacency hypergraphs by 41.2%-44.7%, on the average, for 128-way and 256-way partitions, respectively. For IR data sets, coarsening reduced the size of the boundary adjacency hypergraphs by 96.3%-97.18% on the average. These results imply that coarsening is quite effective in the contraction of the boundary adjacency hypergraphs and provide significant reductions in the size of input supplied to the ILP solver.

4.5 Comparison of Coarsening Algorithms and The Effect of Coarsening

The Dulmage-Mendelsohn decomposition provides quite promising coarsening results. However, it does not take vertex weights and net costs into account. Hence, it is possible that other coarsening algorithms can prove to be more effective than the Dulmage-Mendelsohn decomposition by taking vertex weights and net costs into account. To investigate this issue, we adopted 17 different state-of-the-art coarsening algorithms (HCM, PHCM, MANDIS, AVEDIS, CANBERRA, ABS, GCM, SHCM, HCC, HPC, ABHCC, ABHPC, CONC, GCC, SHCC, NC, MNC) that are implemented in PaToH to obtain coarsened boundary adjacency hypergraphs. We supplied these coarsened boundary adjacency hypergraphs to
the ILP solver and observed their effects on the cutsize reduction.

In our experiments, we evaluated all of the adopted coarsening algorithms over all data sets for different $K$ and $\rho$ settings. We observed that each of the adopted coarsening algorithms show high fluctuations in the quality of the coarsened boundary adjacency hypergraphs. Quality measure in this context is the effectiveness of the ILP phase running on the coarsened hypergraphs. On the other hand, the Dulmage-Mendelsohn decomposition showed a stable performance and in a significant majority (87.6%) of the experiments performed in the top three.

Coarsening provides a lossy compression of the boundary adjacency hypergraph. To further investigate the effectiveness of the coarsening and determine the information loss due to the coarsening, experiments are evaluated with two different setups. In the first setup $S_1$, experiments are evaluated with the coarsening and time limitation constraints. In the second setup $S_2$, ILP phase is performed without coarsening and time limitations. In $S_2$, since there is no limitation on the execution time of the ILP solver, ILP phase dominated the majority of the total runtime in tests. For IR data sets with dense boundary adjacency hypergraphs – e.g., Facebook, Wikipedia – total replication phase took hours to complete. In case of RN data sets, where boundary adjacency hypergraphs are relatively sparse, ILP phase completed in the same amount of time. In $S_2$, ILP phase produced slightly better results in terms of the quality of the replication. In $S_1$, the reduction in the quality due to the loss of information in coarsening varies between 2.3% and 7.8% compared to $S_2$. To sum up, in a majority of the experiments, $S_1$ produces on par results with $S_2$.

### 4.6 Part Visit Orderings

In the conducted experiments, three different part visit ordering schemes are evaluated for hypergraphs given in Table 4.1. In the first scheme $O_1$, parts are ordered by increasing average net degree of their boundary adjacency hypergraph values. In $O_2$, parts are sorted in increasing weight order. In the last scheme $O_3$, 
parts are chosen randomly. On the average, $O_1$ and $O_2$ performs around 5.7-10.3% better results compared to $O_3$ in terms of reduction in the connectivity cutsize. $O_1$ performs slightly (2.2%) better cutsize reductions compared to $O_2$.

To conclude, since ILP phase coupled with coarsening performs quite effective in terms of consuming replication capacity with the maximum possible number of net coverages, part ordering generally causes relatively minor variations in the replication quality. In conducted experiments, results are given according to $O_1$ scheme.

4.7 System Resource Consumptions

At $k$th iteration of the replication algorithm, we construct the boundary adjacency hypergraph $\mathcal{H}_k$, coarsen $\mathcal{H}_k$ to $\mathcal{H}^{\text{coarse}}_k$, select vertices that are to be replicated from $\mathcal{H}^{\text{coarse}}_k$ via ILP. For RN data sets, where boundary adjacency hypergraphs are generally small in size, ILP phase is generally dominated the total runtime of the replication and replication is finished 3-8 times faster than the partitioning time of PaToH. For IR data sets, where boundary adjacency hypergraphs are large in size, 60.8% of the total runtime is consumed by the coarsening, and ILP and $\mathcal{H}_k$ construction took 28.2% and 11.1% of the total runtime, respectively. In the experimented IR data sets, replication of large boundary adjacency hypergraphs performed at most 3.5 times slower compared to the partitioning time of PaToH.
Chapter 5

Conclusion

Motivated by the problem of finding a replication set for a given $K$-way hypergraph partition and a maximum replication capacity ratio, we proposed a part-oriented approach based on a unique blend of an ILP formulation and a coarsening algorithm using the Dulmage-Mendelsohn decomposition. Experiments show that proposed model provides promising results both in terms of the quality of the replication set and the runtime performance. The Dulmage-Mendelsohn decomposition-based coarsening scheme is found to be quite successive for encapsulating the replication characteristics of a hypergraph into its coarsened representation. In the light of conducted experiments, the Dulmage-Mendelsohn decomposition-based coarsening coupled with the ILP formulation provide effective results for covering nets in a boundary adjacency hypergraph.
Appendix A

SUK to MCRS Transformation

In this chapter, we present a simple transformation of SUK (Set-Union Knapsack) problem [42] to an MCRS (Min-Cut Replication Set) problem, which is a generalization of Problem 1 without balancing constraints. SUK problem is defined [42] as follows.

**Definition 4** Set-Union Knapsack Problem. Given a set of $n$ items $N = \{1, 2, \ldots, n\}$ and a set of $m$ so-called elements $P = \{1, 2, \ldots, m\}$, each item $j$ corresponds to a subset $P_j$ of the element set $P$. The items $j$ have nonnegative profits $p_j$, $j = 1, 2, \ldots, n$, and the elements $i$ have nonnegative weights $w_i$, $i = 1, 2, \ldots, m$. The total weight of a set of items is given by the total weight of the elements of the union of the corresponding element sets. Find a subset of the items with total weight not exceeding the knapsack capacity while maximizing the profit.

SUK is known [42] to be an $\mathcal{NP}$-hard problem. A simple transformation of SUK problem to MCRS problem can be given as follows.

**Theorem 1** Every set-union knapsack (SUK) problem can be represented in terms of a min-cut replication set (MCRS) problem.
Proof. One can transform a SUK problem to an MCRS for 2-way hypergraph partition $\Pi = \{V_1, V_2\}$ problem, where elements of SUK problem correspond to the boundary vertices of $V_1$ and element sets correspond to the cut nets. Consider a special MCRS problem where cut nets are connected to a single vertex in $V_2$ whose weight is exceeding the given replication capacity. Thus only replication of vertices in $V_1$ into $V_2$ is possible. A solution to this particular MCRS problem would provide a solution to the SUK problem.

In Fig. A.1, a sample SUK to MCRS transformation is shown. In Fig. A.1a, item set $N$ and element set $P$ are composed of $n$ items and $m$ elements, respectively. Each item $j$ in $N$ is associated with an element set $P_j$, which is a subset of $P$. Objective is to maximize the profit of the covered items, where there is an upper bound on the total weight of the used elements. This SUK instance is mapped to a MCRS problem in Fig. A.1b, where orientation of the replication is forced towards a single direction. That is, the single vertex in part $V_1$ weights much more than the given replication capacity, forcing replication direction from $V_1$ to $V_2$. In addition, items and elements correspond to nets and vertices in Fig. A.1b, respectively. That is, $P_1 = \{1, 2, 3\}$ in Fig. A.1a is represented by net $n_1$ connecting vertices $v_1$, $v_2$, and $v_3$ in Fig. A.1b.
Appendix B

Finding the Cutsizes of a Partition With Vertex Replications

Previous studies involving $K$-way hypergraph partitioning with vertex replications doesn’t investigate the effect of the cutsize metric on the conducted experiments. However, computation of the minimum cutsize for a given partition with vertex replications can stand as a major problem. For instance, the list of cut nets is sufficient to compute the cutsize for cut-net metric (Eq. 2.1). However, pin mapping of the nets (i.e., which partition should be used for a particular pin of a cut net) are necessary for the computation of the cutsize for connectivity metric (Eq. 2.2). Hence, depending on the used cutsize metric, finding the minimum cutsize for a given partition with vertex replications is a significant problem. (Without vertex replications, since every vertex has a unique copy and, hence, every net has a unique pin mapping, this decision problem does not arise.) This issue is generalized in Problem 2.

Problem 2 Finding the Cutsizes of a Partition With Vertex Replication. Given a partition with vertex replication $\Pi_r$ and a cutsize metric $\chi(\cdot)$, find the minimum $\chi(\Pi_r)$. 
Considering the connectivity metric, even for a single net, finding the pin mapping with the least possible number of parts is a set cover optimization problem (i.e., pins correspond to the element universe, and parts correspond to the element sets), which is known to be \( \mathcal{NP} \)-hard. On the other hand, it should be noted that a majority of the pins of a cut net tends to be fixed (i.e., not replicated and has a unique copy in some particular part) and after connecting the floating pins (i.e., pins that can be connected to different copies in different parts) of a cut net to these fixed parts, there remains an insignificant number of pins that needs to be determined for connection. Hence, problem turns out to be relatively cheaply computable in practice. But for a vast number of cut nets, this still can stand as an intractable problem.

For cut-net metric, since Eq. 2.1 just depends on the determination of cut nets, cutsize can be computed in linear time proportional to the size of the total number of pins.
Bibliography


