INDEPENDENT TASK ASSIGNMENT FOR HETEROGENEOUS SYSTEMS

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
E. Kartal Tabak
August, 2013

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

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 dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Enis Çetin

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

Assoc. Prof. Dr. Hakan Ferhatosmanoğlu

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

Asst. Prof. Dr. Ali Aydın Selçuk

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

Asst. Prof. Dr. Engin Demir

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

INDEPENDENT TASK ASSIGNMENT FOR HETEROGENEOUS SYSTEMS

E. Kartal Tabak
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
August, 2013

We study the problem of assigning nonuniform tasks onto heterogeneous systems. We investigate two distinct problems in this context. The first problem is the one-dimensional partitioning of nonuniform workload arrays with optimal load balancing. The second problem is the assignment of nonuniform independent tasks onto heterogeneous systems.

For one-dimensional partitioning of nonuniform workload arrays, we investigate two cases: chain-on-chain partitioning (CCP), where the order of the processors is specified, and chain partitioning (CP), where processor permutation is allowed. We present polynomial time algorithms to solve the CCP problem optimally, while we prove that the CP problem is NP complete. Our empirical studies show that our proposed exact algorithms for the CCP problem produce substantially better results than the state-of-the-art heuristics while the solution times remain comparable.

For the independent task assignment problem, we investigate improving the performance of the well-known and widely used constructive heuristics MinMin, MaxMin and Sufferage. All three heuristics are known to run in $O(KN^2)$ time in assigning N tasks to K processors. In this thesis, we present our work on an algorithmic improvement that asymptotically decreases the running time complexity of MinMin to $O(KN \log N)$ without affecting its solution quality. Furthermore, we combine the newly proposed MinMin algorithm with MaxMin as well as Sufferage, obtaining two hybrid algorithms. The motivation behind the former hybrid algorithm is to address the drawback of MaxMin in solving problem instances with highly skewed cost distributions while also improving the running time performance of MaxMin. The latter hybrid algorithm improves the running time performance of Sufferage without degrading its solution quality. The proposed

algorithms are easy to implement and we illustrate them through detailed pseudocodes. The experimental results over a large number of real-life datasets show that the proposed fast MinMin algorithm and the proposed hybrid algorithms perform significantly better than their traditional counterparts as well as more recent state-of-the-art assignment heuristics. For the large datasets used in the experiments, MinMin, MaxMin, and Sufferage, as well as recent state-of-the-art heuristics, require days, weeks, or even months to produce a solution, whereas all of the proposed algorithms produce solutions within only two or three minutes.

For the independent task assignment problem, we also investigate adopting the multi-level framework which was successfully utilized in several applications including graph and hypergraph partitioning. For the coarsening phase of the multi-level framework, we present an efficient matching algorithm which runs in $O(KN)$ time in most cases. For the uncoarsening phase, we present two refinement algorithms: an efficient $O(KN)$-time move-based refinement and an efficient $O(K^2 N \log N)$-time swap-based refinement. Our results indicate that multi-level approach improves the quality of task assignments, while also improving the running time performance, especially for large datasets.

As a realistic distributed application of the independent task assignment problem, we introduce the site-to-crawler assignment problem, where a large number of geographically distributed web servers are crawled by a multi-site distributed crawling system and the objective is to minimize the duration of the crawl. We show that this problem can be modeled as an independent task assignment problem.

As a solution to the problem, we evaluate a large number of state-of-the-art task assignment heuristics selected from the literature as well as the improved versions and the newly developed multi-level task assignment algorithm. We compare the performance of different approaches through simulations on very large, real-life web datasets. Our results indicate that multi-site web crawling efficiency can be considerably improved using the independent task assignment approach, when compared to relatively easy-to-implement, yet naive baselines.

Keywords: parallel computing, one-dimensional partitioning, load balancing, chain-on-chain partitioning, dynamic programming, parametric search, Parallel processors, heterogeneous systems, independent task assignment, MinMin, MaxMin, Sufferage, constructive heuristics.

ÖZET

HETEROJEN SİSTEMLER İÇİN BAĞIMSIZ İŞ ATAMASI

E. Kartal Tabak
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ağustos, 2013

Bu tezde, heterojen sistemler için büyüklüğü farklılık gösteren işlerin işlemcilere dağıtılması problemleri üzerinde çalıştık. Bu bağlamda iki ayrı problemi inceledik. İlk olarak farklı işlem büyüklüğüne sahip iş katarlarının heterojen işlemcilere bir boyutlu dağıtılması problemi üzerinde durduk. İkinci olarak ise, farklı işlem büyüklüğüne sahip bağımsız işlerin heterojen sistemlerde atanması problemi üzerinde çalıştık.

Farklı büyüklükteki iş katarlarının bir boyutlu parçalanması probleminde iki alt problem üzerinde çalıştık. Birincisi, zincir-zincir parçalama (ZZP) olarak bilenen bir boyutlu sıralı iş zincirinin bir boyutlu sıralı işlemci zinciri üzerine parçalama problemi, ikincisi ise zincir parçalama (ZP) olarak tanımladığımız, bir boyutlu sıralı iş zincirinin sıra önemli olmadan işlemcilere parçalanma problemi. ZZP problemi için heterojen sistemlerde polinom zamanda optimal çözüm bulan algoritmalar sunduk, ZP probleminin ise NP-tam olduğunu ispatladık. Yaptığımız çalışmalarla ZZP probleminde sunduğumuz optimal çözümlerin sezgisel yöntemlerden çok daha iyi sonuçları karşılaştırılabilir sürelerde bulabildiğini ortaya koyduk.

Bağımsız iş atama probleminde, bilinen ve çok kullanılan yapıcı sezgisel algoritmalardan MinMin, MaxMin ve Sufferage algoritmalarının iyileştirilmesi üzerinde çalıştık. Bu sezgisel metotların N işi K işlemciye dağıtırken $O(KN^2)$ zamanda çalıştığı biliniyordu. Bu tezde, MinMin algoritmasında, çözümünü ve çözüm kalitesini değiştirmeden, çalışma zamanını $O(KN \log N)$ zamana düşürecek algoritmik iyileştirmeler yaptık. Ayrıca, MinMin algoritması ile MaxMin ve Sufferage algoritmalarını birleştirerek, iki adet daha hibrit algoritma elde ettik. MaxMin ile MinMin hibritlemesi, MaxMin algoritmasının özellikle kuvvet kanunu gibi özellikleri taşıyan dağılımlardaki dezavantajlarını

gidermenin yanında, MaxMin algoritmasının çalışma hızını da iyileştirdi. Sufferage ile MinMin hibritlemesinin ise Sufferage algoritmasının çözüm kalitesini düşürmeden çalışma hızını iyileştirdi. Algoritmalar için verdiğimiz detaylı akışlar sunduğumuz algoritmaların kolay gerçekleştirilebilir olduklarını göstermektedir. Gerçek hayattan alınan çok sayıdaki örnek veri üzerinde yaptığımız deneyler sunduğumuz MinMin ve hibrit algoritmaların klasik versiyonlarından ve diğer çok kullanılan sezgisel algoritmalardan çok daha iyi çalıştığını gösterdi. Deneylerde kullandığımız büyük örnek veriler için, MinMin, MaxMin ve Sufferage algoritmaları ve diğer çok kullanılan sezgisel algoritmalar günler, haftalar hatta aylar mertebesinde çalışırken, sunduğumuz algoritmalar sonuçları iki-üç dakika içinde hesaplayabildiler.

Bağımsız iş atama probleminde ayrıca, graf ve hipergraf parçalama gibi uygulamalarda başarıyla kullanılmış çok katmanlı mimari yöntemlerinin probleme adaptasyonu üzerinde çalıştık. Çok katmanlı mimarinin katlama aşamasında kullanılmak üzere etkili, çoğu zaman $O(KN)$ sürede çalışan bir algoritma tasarladık. Çok katmanlı mimarinin açma kısmında kullanılmak üzere, iki adet iyileştirme algoritması tasarladık: $O(KN)$ sürede çalışan kaydırma temelli iyileştirme algoritması ve $O(K^2N)$ sürede çalışan değiştirme temelli iyileştirme algoritması. Yaptığımız çalışmalar çok katmanlı yaklaşımların, özellikle büyük örnek veriler için hem iş atama kalitesini hem de çalışma süresi performansını ciddi olarak iyileştirdiğini ortaya koymaktadır.

Bağımsız iş atama probleminin gerçekçi bir dağıtık uygulamasını göstermek üzere, site-indirici eşleştirme problemini inceledik. Bu problem, Internet üzerindeki çok sayıda web sitesinin birden fazla yerde konuşlanmış dağıtık indirici sistemleri vasıtası ile en az sürede tarama işleminin gerceklestirilmesini hedeflemektedir. Bu problemin bağımsız iş atama problemi olarak modellenmesini gerçekleştirdik. Günümüzde kullanılan bağımsız iş atama algoritmalarını sunduğumuz iyileştirilmiş algoritmaları ve çok kaymanlı algoritmamızı problem üzerinde deneyerek karşılaştırdık. Karşılaştırmalarımızda gerçek hayattan alınan çok büyük örnek kümeler kullandık. Sonuçlarımız, kolay gerçekleştirilebilen sezgisel yöntemler yerine, bağımsız iş atama yaklaşımının dağıtık indirici sistemlerin verimliliğini ciddi olarak arttırdığını gösterdi.

Anahtar sözcükler: paralel hesaplama, tek boyutlu parçalama, yük dengeleme, zincir-zincir parçalama, dinamik programlama, parametrik arama, paralel

işlemciler, heterojen sistemler, bağımsız iş yükleme, MinMin, MaxMin, Sufferage, yapıcı sezgisel yöntemler.

Acknowledgement

First I would to thank to my advisor Prof. Cevdet Aykanat. He directed me to the right direction when I lost my way in the academic life. He has spent lengthy nights and lengthy years with me. Without his patience and support, I would not be the one that I am at the moment.

I would like to thank the faculty of Computer Engineering Department. They are all supportive, and ready to help when I needed.

I would like to thank all my friends in Bilkent University, and especially the old and new members of Parallel Computing Group. Special thanks to Berkant Barla Cambazoğlu.

I cannot forget my collegues at HAVELSAN A.Ş. I would like to thank them all.

I'd like to thank also my son, for being my son, and for letting me share less time with him.

And most of all, I would like to thank my wife for her deepless love, encouragement and patience. She always supported me in all possible ways to pursue this PhD title.

To my wife and my son

Contents

1 Introduction **1**

2 Related Work **6**

2.1 1D chains-on-chains partitioning 6

2.2 Independent Task Assignment 9

2.3 Multi-Level Framework 11

3 One-Dimensional Partitioning for Heterogeneous Systems: Theory and Practice **13**

3.1 Chain-on-chain (CCP) Problem for Heterogeneous Systems 15

3.2 CCP Algorithms for Homogenous Systems 16

3.2.1 Heuristics 16

3.2.2 Dynamic Programming 17

3.2.3 Parametric Search 17

3.3 Proposed CCP Algorithms for Heterogeneous Systems 19

3.3.1 Heuristics 20

3.3.2 Dynamic Programming 23

3.3.3 Parametric Search 27

3.4 Chain Partitioning (CP) Problem for Heterogeneous Systems . . . 33

3.5 Experimental Results 35

3.5.1 Experimental Setup 35

3.5.2 CCP Experiments 39

3.5.3 CP Experiments 47

4 Independent Task Assignment: Improving Performances of Well-known Constructive Heuristics 50

4.1 Existing Algorithms 50

4.2 MinMin+ 57

4.3 MaxMin+ 60

4.4 Suff+ 68

4.5 GA+ 69

4.6 Experimental Results 71

4.6.1 Datasets 71

4.6.2 Performance Analysis 74

5 Geographically Distributed Web Crawling: A Task Assignment Approach 88

5.1 Web Crawling and Independent Task Assignment 88

5.2 Server-to-Crawler Assignment Problem 92
5.2.1 Architecture 92
5.2.2 Notation 94
5.2.3 Problem 95
5.3 Heuristic Solutions 97
5.4 Cost Model for Crawling Times 100
5.5 Experiments 103
5.5.1 Datasets 103
5.5.2 Centralized Crawling Performance 104
5.5.3 Performance of Task Assignment Heuristics 107
5.6 Related Work on Web Crawling 111

6 Independent Task Assignment of Very Large Sets: A Multi-Level Approach 113

6.1 Coarsening 114
6.1.1 Dissimilarity Metric for Matching 115
6.1.2 Efficient Matching Algorithm 116
6.2 Initial Task-to-Processor Assignment 120
6.3 Uncoarsening 120
6.3.1 Move Refinement 121
6.3.2 Swap Refinement 124

6.4 Experiments 127

7 Conclusion 134

A Detailed Analysis 151

A.1 Average Case Analysis of NICOL+ 151

B Code 154

C The Process of Generation of ETC Matrices for ClueWeb-09 datasets 155

C.1 Outline 155

C.2 Obtain the Crawl Data 156

C.3 WarcProcessor1 156

C.4 WarcProcessor5 158

C.5 WarcProcessor5B 159

C.6 WarcProcessor2 159

C.7 WarcProcessor1B 161

C.8 WarcProcessor1c 161

C.9 WarcProcessor3 162

C.10 Ip2Geo 164

C.11 WarcProcessor6 165

C.12 WarcProcessor7 165

C.13 WarcProcessor7B 167

List of Figures

3.1 Visualization of direct volume rendering dataset workloads. Top: workload distributions of 2D task arrays. Bottom: histograms showing weight distributions of 1D task chains. 36

3.2 Visualization of sparse matrix dataset workloads. Left: non-zero distributions of the sparse matrices. Right: histograms showing weight distributions of the 1D task chains. 37

4.1 Log-log plots of the cumulative density distribution of task weights for skewed datasets. x-axis: weights of tasks, y-axis: cumulative density distribution, i.e., $P(X \geq x)$. 72

4.2 Log-log plots of the cumulative density distribution of task weights for non-skewed datasets. x-axis: weights of tasks, y-axis: cumulative density distribution, i.e., $P(X \geq x)$. 73

5.1 A geographically distributed web crawling architecture with four data centers (DC1–DC4), each crawling a non-overlapping subset of web servers (the circles in the figure) in the Web. 93

5.2 The cumulative density distribution for the number of pages on servers. x-axis (log scale): number of pages on servers, y-axis (log scale): cumulative density distribution, i.e., $P(X \geq x)$. 103

5.3 The cumulative density distribution for the amount of content (in bytes) on servers. x-axis (log scale): amount of content on servers, y-axis (log scale): cumulative density distribution, i.e., $P(X \geq x)$. 104

5.4 Log-log plots of the cumulative density distribution of best, average, and worst crawler download times of servers in seconds. x-axis: download times, y-axis: cumulative density distribution, i.e., $P(X \geq x)$. 106

C.1 WarcProcessor1 data flow 157

C.2 WarcProcessor5 data flow diagram. 158

C.3 WarcProcessor5B data flow diagram. 159

C.4 WarcProcessor2 data flow diagram. WarcProcessor2 merges host files into a single sorted hosts file 160

C.5 WarcProcessor1B data flow diagram. 161

C.6 WarcProcessor1C data flow diagram. 162

C.7 WarcProcessor3 data flow diagram. 163

C.8 WarcProcessor3 internal data flow diagram. 163

C.9 Ip2Geo data flow diagram. 165

C.10 WarcProcessor6 data flow diagram. 166

C.11 WarcProcessor7 data flow diagram. 166

C.12 WarcProcessor7B data flow diagram. 167

List of Tables

3.1 The summary of important abbreviations and symbols used in this chapter 14

3.2 Properties of the test set 38

3.3 Percent load imbalance values for the processor speed range of 1–8 for the volume rendering dataset 39

3.4 Percent load imbalance values for the processor speed range of 1–8 for the sparse matrix dataset 40

3.5 Percent load imbalance values for different processor speed ranges for the volume rendering dataset 41

3.6 Percent load imbalance values for different processor speed ranges for the sparse matrix dataset 42

3.7 Partitioning times (in msecs) for the processor speed range of 1–8 for the volume rendering dataset 44

3.8 Partitioning times (in msecs) for the processor speed range of 1–8 for the sparse matrix dataset 45

3.9 Partitioning time averages (over K) of the exact CCP algorithms normalized with respect to those of the RB heuristic for different processor speed ranges 46

3.10 Geometric averages (over K) of percent load imbalance values for R randomly ordered processor chains for the volume rendering dataset with the processor speed range of 1–8 47

3.11 Geometric averages (over K) of percent load imbalance values for R randomly ordered processor chains for the sparse matrix dataset with the processor speed range of 1–8 48

3.12 Best percent load imbalance values for $R = 10000$ randomly ordered processor chains with different processor speed ranges . . . 49

4.1 The notation used in this chapter 51

4.2 Properties of the datasets 70

4.3 Percent load imbalance values for social network datasets 75

4.4 Percent load imbalance values for distributed web crawling datasets 75

4.5 Percent load imbalance values for parallel DVR datasets 75

4.6 Percent load imbalance values for parallel SpMxV datasets 76

4.7 Percent load imbalance values for parallel SpMxV datasets (2) . . 77

4.8 Averages of percent load imbalance values over all datasets 78

4.9 Running times (seconds) of heuristics for social network datasets . 79

4.10 Running times (seconds) of heuristics for distributed web crawling datasets 79

4.11 Running times (seconds) of heuristics for parallel DVR datasets . 80

4.12 Running times (seconds) of heuristics for parallel SpMxV datasets 81

4.13 Running times (seconds) of heuristics for parallel SpMxV datasets (2) 82

4.14 Normalized running time averages over all datasets 82

4.15 Number of MaxMin-based assignments performed by MaxMin+ for social network, distributed web crawling and parallel DVR datasets 83

4.16 Number of MaxMin-based selections performed by MaxMin+ for parallel SpMxV datasets 84

5.1 The notation used in this chapter 91

5.2 Properties of the datasets 103

5.3 Distribution of web content on countries 105

5.4 Performance of a centralized crawler located in a particular country 106

5.5 Percent load imbalance values 109

5.6 Expected crawling duration (in days) 110

5.7 Execution times (in seconds) 111

6.1 Properties of the datasets 127

6.2 Percent load imbalance values 131

6.3 Execution times (in seconds) 132

Chapter 1

Introduction

In many applications of parallel and distributed computing, load balancing is achieved by static assignment at a preprocessing step. Static task assignment at the preprocessing step is a crucial component in the efficiency of the parallel and distributed applications. In this thesis, we will describe efficient solutions to several heterogeneous partitioning and task assignment algorithms.

The main problems in focus of this thesis are: one-dimensional (1D) heterogeneous partitioning and heterogeneous independent task assignment problems.

In Section 2.1, we describe the background for the homogenous 1D partitioning problem and the techniques to solve the well-known homogenous 1D CCP (chains-on-chains) problem. In Chapter 3, we investigate how these techniques can be generalized for heterogeneous systems, where processors have varying computational powers. Two distinct problems arise in partitioning chains for heterogeneous systems. The first problem is the CCP problem, where a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N tasks with associated computational weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$ is to be mapped onto a chain of $\mathcal{P} = \langle P_1, P_2, \ldots, P_K \rangle$ of K processors, i.e., the pth task subchain in a partition is assigned to the pth processor. The execution time of task t_i on processor P_p is w_i/e_p. The objective is to minimize the completion time of the latest finishing task. The second problem is the chain partitioning (CP) problem, where a chain of tasks is to be

mapped onto a *set*, as opposed to a chain, of processors, i.e., processors can be permuted for subchain assignments. For brevity, the CCP problem for homogenous systems and heterogeneous systems will be referred to as the homogenous CCP problem and heterogeneous CCP problem, respectively. The CP problem refers to the chain partitioning problem for heterogeneous systems, since it has no counterpart for homogenous systems.

In Chapter 3, we show that the heterogeneous CCP problem can be solved in polynomial time by enhancing the exact algorithms proposed for the solution of the homogenous CCP problem [93]. We present how these exact algorithms for homogenous systems can be enhanced for heterogeneous systems and implemented efficiently for runtime performance. We also present how the heuristics widely used for the solution of homogenous CCP problem can be adapted for heterogeneous systems. We present the implementation details and pseudocodes for the exact algorithms and heuristics for clarity and reproducibility. Our experiments with workload arrays coming from image-space-parallel volume rendering and row-parallel sparse matrix vector multiplication applications show that our proposed exact algorithms produce substantially better results than the heuristics while the partitioning times remain comparable. On average, optimal solutions provide 4.9 and 8.7 times better load imbalance than heuristics for 128-way partitionings of volume rendering and sparse matrix datasets, respectively. On average, the time it takes to compute an optimal solution is less than 2.20 times the time it takes to compute an approximation using heuristics for 128 processors, and thus the preprocessing times can be easily compensated by the improved efficiency of the subsequent computation even for a few iterations.

The CP problem on the other hand, is NP-complete as we prove in Chapter 3. Our proof uses a pseudo-polynomial reduction from the 3-Partition problem, which is known to be NP-complete in the strong sense [50]. Our empirical studies showed that processor ordering has a very limited effect on the solution quality, and an optimal CCP solution on a random processing ordering serves as an effective CP heuristic.

Another important focus of this work is on the independent task assignment

problem, which often arises in applications related to heterogeneous computing systems. In this problem, we have a set $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$ of N independent tasks, a set $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ of K heterogeneous processors, and an expected-time-to-compute matrix $E = \{x_{i,k}\}_{N \times K}$, where $x_{i,k}$ denotes the expected execution cost of task T_i on processor P_k. The objective is to find a task-to-processor assignment that results in the minimum turnaround time (makespan). In other words, the objective is to minimize the load of the maximally loaded (bottleneck) processor. This problem is known to be NP-complete [63].

One of the most popular heuristics used for solving the independent task assignment problem is the `MinMin` heuristic. It is constructive, simple, and is reported to produce high quality assignments. However its running time is reported to be $O(KN^2)$, which prevents the algorithm to be used on problem instances with large number of tasks. We believe that the computational complexity of `MinMin` is overlooked in the parallel and distributed computing literature. This mainly stems from the task-oriented view of `MinMin`, constituting a lower bound of $\Omega(KN^2)$ on the running time. In this thesis, we propose an $O(KN \log N)$-time algorithm that improves this quadratic lower bound by switching from the task-oriented view to a processor-oriented view. The proposed `MinMin` algorithm, which is referred to herein as `MinMin+`, attains exactly the same solution quality as `MinMin` without degrading the ease of implementation. The results of our experiments over a wide range of problem instances indicate that `MinMin+` runs several orders of magnitude faster than `MinMin`. For a large dataset that contains about 2.5 million tasks, `MinMin` finds a 16-way assignment in about 22 days, whereas `MinMin+` finds the same assignment in about a minute.

Two other well-known constructive heuristics used for solving the independent task assignment problem are MaxMin (`MaxMin`) [5, 10, 48, 63] and Sufferage (`Suff`) [78]. These heuristics differ from `MinMin` in the task selection policy adopted during the task-to-processor assignment process. In Chapter 4, we propose improvements over these two heuristics as well. We combine `MaxMin` with `MinMin+` as well as `Suff` with `MinMin+` to obtain the hybrid algorithms `MaxMin+` and `Suff+`, respectively.

The assignment of large tasks to their favorite processors[1] is important to obtain a good makespan, especially in skewed datasets. Although the `MaxMin` heuristic assigns the largest task to its favorite processor, its inherent mechanism is likely to fail to assign remaining large tasks to their favorite processors. The motivation behind `MaxMin+` is to address this drawback of `MaxMin` in solving problem instances with highly skewed cost distributions while also improving the running time performance of `MaxMin`.

`Suff` is reported to be among the algorithms that yield high-quality solutions [69, 78, 107]. Despite its success, the quadratic running time prevents the application of this heuristic to large datasets. The motivation behind `Suff+` is to improve the running time performance of `Suff` without degrading the solution quality.

Although both `MaxMin+` and `Suff+` are, in the worst case, still $O(KN^2)$-time algorithms, our experimental results show that they run considerably faster than the traditional `MaxMin` and `Suff` heuristics, respectively. The experimental results also indicate that `MaxMin+` finds considerably better solutions than `MaxMin` while `Suff+` finds slightly better solutions than `Suff`, on average.

`MinMin` is also used as a component in the design of more complex algorithms [10, 105, 108]. Genetic algorithm (`GA`) [10, 105] is a typical example of such complex algorithms. In this work, we also demonstrate that the running time performance of the `GA` algorithm can be significantly improved simply by replacing `MinMin` with `MinMin+`, without affecting the original solution quality at all.

We also investigate adopting the multi-level framework which is successfully utilized in several applications including graph and hypergraph partitioning [13, 20, 67]. In Section 2.3, we describe the background on multi-level framework. In Chapter 6, we describe our proposed multi-level algorithm for the independent task assignment problem. Multi-level algorithms execute in three phases: coarsening, initial solution, and uncoarsening. In the coarsening phase, the problem instance is coarsened to a smaller problem instance. For the coarsening phase

[1] A processor P_k is said to be a favorite processor for a task T_i if the expected cost of T_i is minimum on P_k, i.e., $k = \text{argmin}_\ell \, x_{i,\ell}$.

of the multi-level framework, we present an efficient matching algorithm which runs in $O(KN)$ time in most cases. Initial solution phase finds an initial solution at the coarsest problem instance. In the uncoarsening phase, problem instance is uncoarsened and refined at the finer level. For the uncoarsening phase, we present two novel refinement algorithms: an efficient $O(KN)$-time move-based refinement and an efficient $O(K^2N \log N)$-time swap-based refinement algorithms. Our results indicate that multi-level approach improves the quality of task assignments, while also improving the running time performance, especially for large datasets.

We also demostrate the improved solutions to the independent task assignment problem on a very large scale real-life problem of distributed web crawling. So far, independent task assignment is not being used in the domain of distributed web crawling. Possibly the large asymptotic runtime complexities of good heuristics prevented them to be used on task asssignment problems of web crawling, which have very large number of tasks. We show that the assignment problem of distributed web crawling can be formulated as a task assignment problem. Regarding that topic, we make the following contributions. We introduce two variants of the task assignment problem for geographically distributed web crawling architectures. We adapt several task assignment algorithms taken from the literature to one of these problems. We conduct experiments using real-life web data collections and network statistics. The obtained results demonstrate the potential performance improvements that can be attained by our approach over relatively easy-to-implement but naive baseline approaches.

The outline of this thesis is as follows: Chapter 2 describes the background and related work of our target problems. Chapter 3 describes the heteregeneous 1D CCP partitioning problem and efficient exact algorithms which are guaranteed to find a globally optimum solution. Chapter 4 describes novel asympthotical and practical improvements on well-known independent task assignment heuristics. Chapter 5 presents our adoptation of assignment problem in distributed web crawling as an independent task assignment problem. Chapter 6 describes a multi-level approach for very-large-scale independent task assignment problem instances. We finally conclude with Chapter 7.

Chapter 2

Related Work

In this chapter, we provide a discussion on previous work on our target assignment problems. Section 2.1 discusses the related work on one-dimensional (1D) chains-on-chains (CCP) and Section 2.2 discusses related work on the independent task assignment algorithms. Section 2.3 provides a discussion on the related work on multi-level framework.

2.1 1D chains-on-chains partitioning

In many applications of parallel computing, load balancing is achieved by mapping a possibly multi-dimensional computational domain down to a one-dimensional (1D) array, and then partitioning this array into parts with equal weights. Space filling curves are commonly used to map the higher dimensional domain to a 1D workload array to preserve locality and minimize communication overhead after partitioning [21, 37, 72, 91]. Similarly, processors can be mapped to a 1D array so that communication is relatively faster between close processors in this processor chain [74]. This eases mapping for computational domains and improves efficiency of applications. The load balancing problem for these applications can be modeled as the chain-on-chain partitioning (CCP) problem, where we map a chain of tasks onto a chain of processors. Formally, the objective of

the CCP problem is to find a sequence of $K-1$ separators to divide a chain of N tasks with associated computational weights into K consecutive parts to minimize maximum load among processors.

More formally, in the homogenous CCP problem, a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N tasks with associated *positive* computational weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$ is to be mapped onto an identical processor chain of K processors. A *task subchain* $\mathcal{T}_{i,j} = \langle t_i, t_{i+1}, \ldots, t_j \rangle$ is defined as a subset of contiguous tasks. The computational weight of $\mathcal{T}_{i,j}$ is $W_{i,j} = \sum_{i \leq h \leq j} w_h$. In the homogenous CCP, a partition Π should map contiguous task subchains to contiguous processors. Hence, a K-way partition of a task chain with N tasks onto a processor chain with K processors is described by a sequence $\Pi = \langle s_0, s_1, \ldots, s_K \rangle$ of $K+1$ separator indices, where $s_0 = 0 \leq s_1 \leq \cdots \leq s_K = N$. Here, s_p denotes the index of the last task of the pth part so that pth processor P_p receives the task subchain $\mathcal{T}_{s_{p-1}+1, s_p}$ with load $W_{s_{p-1}+1, s_p}$. The cost $C(\Pi)$ of a partition Π is determined by the maximum processor load among all processors, i.e.,

$$C(\Pi) = \max_{1 \leq p \leq K} \left\{ W_{s_{p-1}+1, s_p} \right\} \tag{2.1}$$

The objective of CCP problem is to find a partition Π_{opt} that minimizes the bottleneck value $C(\Pi_{\text{opt}})$.

A solution to the CCP problem is first proposed by Bokhari [8]. Bokhari's algorithm runs in $O(N^3K)$ time, which is based on finding a minimum path on a layered graph. Nicol and O'Hallaron [85] presented an $O(N^2K)$-time algorithm by decreasing the number of edges on Bokhari's graph. Following studies on homogenous CCP can be categorized as: dynamic programming, iterative refinement, and parametric search.

Dynamic-programming approach is initiated with $O(N^2K)$-time algorithms independently by Anily and Federgruen [3] and Hansen and Lih [55]. Choi and Narahari [28] and Manne and Olstad [79] proposed faster algorithms and reduced the time complexity of dynamic-programming approach to $O(NK)$ and $O((N-K)K)$, respectively. Pinar and Aykanat proposed a dynamic programming algorithm with an $O(N + K \log N + K^2 w_{\max}/w_{\text{avg}})$ time complexity, where $w_{\max}$ is the load of the maximum weighted task and w_{avg} is the average load of

all tasks. Their algorithm becomes linear in N when $w_{\max} = O(W_{\text{tot}}/K^2)$ and $N \gg K$.

In the iterative refinement approach, the algorithms start with an initial solution, and the solution is iteratively improved. Manne and Sørevik proposed an iterative refinement approach with an $O((N-K)K \log K)$-time algorithm. Pinar and Aykanat [93] proposed an iterative refinement algorithm which has a runtime complexity of $O(N + K \log N + K^3(w_{\max}/w_{\text{avg}}))$ in the worst case.

The parametric search approach is based on a probing function which succeeds or fails for a given candidate bottleneck value. The runtime complexity of the probing function is $\theta(N)$, because each task has to be examined. However, when there will be lots of calls to the probing function, the function can utilize an index structure to reduce the complexity. Iqbal's prefix-sum operation [64] on the task chain can be used as an index structure. The prefix sum can be implemented in $O(N)$ time, and probing can be implemented in $O(K \log N)$ through binary search on the prefix-summed array. Later, Han et al. [54] reduced the complexity of the probing function to $O(K \log N/K)$.

Parametric search starts with Iqbal's ϵ-approximation algorithm. It performs $O(\log W_{\text{tot}}/\epsilon)$ probe calls, where W_{tot} denotes the total task weight. Iqbal's algorithm is a result of the observation that the bottleneck value is between W_{tot}/K and W_{tot}. Iqbal followed a binary search on the bottleneck values within that region. Nicol and O'Hallaron proposed an optimal parametric search algorithm that performs at most $4N$ probes [85, 86]. This algorithm has restrictions on task weights. Iqbal and Bokhari later relaxed those restrictions on Nicol's proposal [66]. Iqbal [65] and Nicol [84] independently proposed another search scheme that for an optimal solution that requires only $O(K \log N)$ probe calls. Pinar and Aykanat [93] proposed two optimal parametric search algorithms, the first algorithm has an expected runtime complexity of $O(N + K \log K \log N + K \log N \log w_{\max}/w_{\text{avg}})$. Their second algorithm is an improvement of Nicol's algorithm [84] and has a runtime complexity of $O(N + K \log N + w_{\max}(K \log K)^2 + w_{\max}K^2 \log K \log (w_{\max}/w_{\text{avg}}))$.

Most of these efforts are for the optimal solution to the homogenous CCP

problem. Despite those, heuristics are also used. [81] is an example to heuristics. Heuristics may be preferred because of their ease of implementation, their efficiency, and some additional specific characteristics such as parallelizability. The heuristics are reported to be unnecessary with the existence of simple and efficient optimal solutions [93].

Asymptotically efficient algorithms exists for the solution of homogenous CCP problem. Frederickson [46, 47] proposed an $O(N)$-time algorithm using partitioning trees. Han et al. proposed a recursive algorithm with a complexity of $O(N + K^{1+\epsilon})$ for any $\epsilon > 0$. Although these algorithms are asymptotically efficient, they are reported to be impractical [93].

2.2 Independent Task Assignment

Assigning a set of independent tasks to a set of nonidentical processors is a common problem which often arises in parallel and distributed systems. The problem is known in the literature as *independent task assignment* problem. Formally, the objective of the independent task assignment problem is to assign N independent tasks to K heterogeneous processors such that turnaround time (makespan, load of maximally loaded processor) is minimized.

More formally, in the independent task assignment problem, we have a set $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$ of N independent tasks, a set $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ of K heterogeneous processors, and an expected-time-to-compute matrix $E = \{x_{i,k}\}_{N\times K}$, where $x_{i,k}$ denotes the expected execution cost of task T_i on processor P_k. In independent task assignment, an assignment should assign tasks to processors. Hence, an assignment is described by a vector $A = \{a_i\}_N$ of N elements. Here, a_i denotes the assignment of task T_i to processor P_{a_i}. The makespan $M(A)$ of this assignment is determined by the maximum processor load among all processors:

$$M(A) = \max_{1\leq k\leq K} \left\{ \sum_{\{i|a_i=k\}} \{x_i, k\} \right\} \quad (2.2)$$

The objective of independent task assignment is to find an assignment vector

A_{OPT} that minimizes the makespan value $M_{\text{OPT}} = M(A_{\text{OPT}})$. The problem is NP-complete [12, 43, 59, 63].

Independent task assignment problem is first introduced by Bruno et al. [12]. They also proposed the first heuristics to the independent task assignment problem. However, in their work, their main concern is to minimize mean finish times of tasks and they discussed independent task assignment problem as a supporting side topic. Horowitz and Sahni [59] provided an exact solution for the special integer-weighted ETC-matrix case of the problem with exponential complexity of $O(K^N)$. Horowitz and Sahni also described an approximation algorithm with complexity $O((1/\epsilon)N^{2K})$, where ϵ is the distance from optimal solution. Ibarra and Kim [63] are the first to introduce practical heuristics for the independent task assignment problem. The well-known `MinMin` and `MaxMin` are the heuristics introduced by this work. Since then, the area attracted many researchers to propose new and better heuristics to the independent task assignment problem or to use the solutions in other possibly more complex algorithms [5,10,24,29,32,42,48,69,77,78,90,94,95,101,104,105,107,108]. Maheswaran et al. [78] introduced the `Suff` heuristic. Braun et al. [10] evaluated 11 common heuristics for the problem and reported that `MinMin` is best on their testbed. Luo et al. [77] presented 20 heuristics grouped under an hierarchy.

The `MinMin` heuristic is first introduced in [63] and since then it is used many times for solving the independent task assignment problem [5, 10, 23, 35, 39, 63, 68, 75, 77, 78, 89, 97, 103]. `MinMin` is a constructive heuristic with some desirable properties. It is free of parameters that require tuning and is easy to implement. Moreover, it is reported to produce “high quality” solutions. Since its first proposal, the running time of the `MinMin` algorithm is reported to be $O(KN^2)$ in the literature [5, 23, 39, 63, 68, 75, 77, 78, 89]. Despite its success, the quadratic running time complexity of the heuristic prevents its use in problem instances where the number of tasks to be assigned is very large. Recently, the `MinMin` algorithm is parallelized to enable the application of the algorithm to large datasets [94]. This parallel version runs in $O(N^2K/P + N^2 + N \log P)$ time, where P denotes the number of homogenous processors used in parallelization of the `MinMin` algorithm (P may be different than K).

2.3 Multi-Level Framework

Multi-level framework is a widely used pattern in applying iterative improvement refinements, especially in graph and hypergraph partitioning. In K-way graph (hypergraph) partitioning problem, given a graph (hypergraph), the problem is to find a partition of the vertices into K roughly equal disjoint subsets such that the number of edges (hyperedges) connecting different subsets is minimized, while maintaining balancing contraints. Objectives and constraints slightly change in different versions of the problems. The problem is NP-hard, and several heuristics have been developed. Kernighan and Lin [70] proposed the famous KL algorithm, which is an iterative improvement heuristic for graph partitioning. The algorithm is applied to hypergraph partitioning by Schweikert and Kernighan [98]. Fiduccia and Mattheyses [45] introduced a faster implementation of the KL algorithm for hypergraph partitioning, which is the famous FM algorithm.

Although FM algorithm is fast and can produce good solutions, the performance of the FM algorithm detoriates for large and sparse graphs/hypergraphs. Moreover, the quality of FM algorithm is not stable: on average the solutions generated by FM is worse than the solution of the KL. Running the FM algorithm several times from random initial partitionings and picking the best solution is a proposal to alleviate the problem [2]. Two-phase algorithms are introduced to overcome the deficiencies [52]. In this version, a clustering algorithm is applied to the original problem instance to obtain a coarser problem instance. Clustering is performed on highly connected vertices. Then FM is executed on the coarser instance and the resulting solution is projected back to the original problem instance. FM is executed again at this level. Several algorithms are proposed for better clustering and refinement for two-phase framework [33, 99].

The two-phase approach is then extended to multi-level approach [13, 20, 57, 67]. The multi-level approach consists of three phases: Coarsening, initial solution, and uncoarsening. In the coarsening phase, a multi-level clustering is applied starting from the original graph/hypergraph by adopting various matching heuristics until the size of the coarsened graph reduces below a predetermined threshold value. In the initial solution phase, the coarsest graph is partitioned

using various heuristics. In the uncoarsening phase, the partition found in the initial solution is successively projected back towards the original graph/hypergraph by refining the projected partitions on the intermediate levels using an iterative improvement algorithm. The success of multi-level approach both in runtime and solution quality makes it a standard for the graph and partitioning problem.

Chapter 3

One-Dimensional Partitioning for Heterogeneous Systems: Theory and Practice

In this chapter, we describe our studies on 1D heterogeneous CCP problem. The sections of this chapter is organized as follows. Table 3.1 summarizes important symbols used throughout the chapter. Section 3.1 introduces the heterogeneous CCP problem. In Section 3.2, we summarize the solution methods for homogenous CCP. In Section 3.3, we discuss how solution methods for homogenous systems can be enhanced to solve the heterogeneous CCP problem. In Section 3.4, we discuss the CP problem, prove that it is NP-Complete. In Section 3.5, we present our experiment results.

Table 3.1: The summary of important abbreviations and symbols used in this chapter

Notation	Explanation
N	number of tasks
$\mathcal{T}$	task chain, i.e., $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$
t_i	ith task in the task chain
$\mathcal{T}_{i,j}$	task subchain of tasks from t_i upto t_j, i.e., $\mathcal{T}_{i,j} = \langle t_i, t_{i+1}, \ldots, t_j \rangle$
w_i	computational load of task t_i
$w_{\max}$	maximum computational load among all tasks
w_{avg}	average computational load of all tasks
$w_{\min}$	minimum computational load of all tasks
$W_{i,j}$	total computational load of task subchain $\mathcal{T}_{i,j}$
W_{tot}	total computational load, i.e., $W_{\text{tot}} = W_{1,N}$
K	number of processors
$\mathcal{P}$	processor chain, i.e., $\mathcal{P} = \langle P_1, P_2, \ldots, P_K \rangle$ in the CCP problem
	processor set, i.e., $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ in the CP problem
P_p	pth processor in the processor chain
$\mathcal{P}_{q,r}$	processor subchain from P_q upto P_r, i.e., $\mathcal{P}_{q,r} = \langle P_q, P_{q+1}, \ldots, P_r \rangle$
e_p	execution speed of processor P_p
$E_{q,r}$	total execution speed of processor subchain $\mathcal{P}_{q,r}$
E_{tot}	total execution speed of all processors, i.e., $E_{\text{tot}} = E_{1,K}$
B^*	ideal bottleneck value
UB	upper bound on the value of an optimal solution
LB	lower bound on the value of an optimal solution
s_p	index of the last task assigned to the pth processor.
$\lg x$	base-2 logarithm of x, i.e., $\lg x = \log_2 x$.

3.1 Chain-on-chain (CCP) Problem for Heterogeneous Systems

In the heterogeneous CCP problem, a computational problem, which is decomposed into a chain $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N tasks with associated *positive* computational weights $\mathcal{W} = \langle w_1, w_2, \ldots, w_N \rangle$ is to be mapped onto a processor chain $\mathcal{P} = \langle P_1, P_2, \ldots, P_K \rangle$ of K processors with associated execution speeds $\mathcal{E} = \langle e_1, e_2, \ldots, e_K \rangle$. The execution time of task t_i on processor P_p is w_i/e_p. For clarity, we note that there are no precedence constraints among the tasks in the chain.

A *task subchain* $\mathcal{T}_{i,j} = \langle t_i, t_{i+1}, \ldots, t_j \rangle$ is defined as a subset of contiguous tasks. Note that $\mathcal{T}_{i,j}$ defines an empty task subchain when $i > j$. The computational weight of $\mathcal{T}_{i,j}$ is $W_{i,j} = \sum_{i \le h \le j} w_h$. A partition Π should map contiguous task subchains to contiguous processors. Hence, a K-way partition of a task chain with N tasks onto a processor chain with K processors is described by a sequence $\Pi = \langle s_0, s_1, \ldots, s_K \rangle$ of $K + 1$ separator indices, where $s_0 = 0 \le s_1 \le \cdots \le s_K = N$. Here, s_p denotes the index of the last task of the pth part so that processor P_p receives the task subchain $\mathcal{T}_{s_{p-1}+1, s_p}$ with load $W_{s_{p-1}+1, s_p}/e_p$. The cost $C(\Pi)$ of a partition Π is determined by the maximum processor load among all processors, i.e.,

$$C(\Pi) = \max_{1 \le p \le K} \left\{ \frac{W_{s_{p-1}+1, s_p}}{e_p} \right\} \tag{3.1}$$

This $C(\Pi)$ value of a partition is called its *bottleneck value*, and the processor defining it is called the *bottleneck processor*. The CCP problem is to find a partition Π_{opt} that minimizes the bottleneck value $C(\Pi_{\text{opt}})$.

Similar to the task subchain, a processor subchain $\mathcal{P}_{q,r} = \langle P_q, P_{q+1}, \ldots, P_r \rangle$ is defined as a subset of contiguous processors. Note that $\mathcal{P}_{q,r}$ defines an empty processor subchain when $q > r$. The computational speed of $\mathcal{P}_{q,r}$ is $E_{q,r} = \sum_{q \le p \le r} e_p$.

The ideal bottleneck value B^* is defined as

$$B^* = \frac{W_{\text{tot}}}{E_{\text{tot}}}, \tag{3.2}$$

where E_{tot} is the sum of all processor speeds and W_{tot} is the total task weight; i.e., $E_{\text{tot}} = E_{1,K}$ and $W_{\text{tot}} = W_{1,N}$. Note that B^* can only be achieved when all processors are equally loaded, so it constitutes a lower bound on the achievable bottleneck values, i.e., $B^* \le C(\Pi_{\text{opt}})$.

3.2 CCP Algorithms for Homogenous Systems

The homogenous CCP problem can be considered as a special case of the heterogeneous CCP problem, where the processors are assumed to have equal speed, i.e., $e_p = 1$ for all p. In this section, we restate the existing heuristics for homogenous systems, which we will adopt for heterogeneous systems in Section 3.3.

3.2.1 Heuristics

Possibly the most commonly used CCP heuristic is *recursive bisection* (RB), a greedy algorithm. RB achieves P-way partitioning through $\lg P$ levels of bisection steps. At each level, the workload array is divided evenly into two. RB finds the optimal bisection at each level, but the sequence of optimal bisections at each level may lead to a multi-way partition which is far away from an optimal. Pınar and Aykanat [93] proved that RB produces partitions with bottleneck values no greater than $B^* + w_{\max}(K-1)/K$.

Miguet and Pierson [81] proposed another heuristic that determines s_p by bi-partitioning the task chain in proportion to the length of the respective processor subchains. That is, s_p is selected in such a way that $W_{1,s_p}/W_{1,N}$ is as close to the ratio p/K as possible. Miguet and Pierson [81] prove that the bottleneck value found by this heuristic has an upper bound of $B^* + w_{\max}$.

These heuristics can be implemented in $O(N + K \log N)$ time. The $O(N)$

time is due to prefix-sum operation on the tasks array, after which each separator index can be found by a binary search on the prefix-summed array.

3.2.2 Dynamic Programming

The overlapping subproblems and the optimal substructure properties of the CCP problem enable dynamic programming solutions. The overlapping subproblems are partitioning the first i tasks onto the first p processors, for all possible i and p values. For the *optimal substructure* property, observe that if the last processor is not the bottleneck processor in an optimal partition, then the partitioning of the remaining tasks onto the first $K-1$ processors must be optimal. Hence, the recursive definition for the bottleneck value of an optimal partition is

$$B_i^p = \min_{0 \le j \le i} \left\{ \max \left\{ B_j^{p-1}, W_{j+1,i} \right\} \right\} \tag{3.3}$$

Here, B_i^p denotes the optimal solution value for partitioning the first i tasks onto the first p processors. In Eq. (3.3), searching for index j corresponds to searching for separator s_{p-1} so that the remaining subchain $\mathcal{T}_{j+1,i}$ is assigned to the last processor in an optimal partition. This definition defines a dynamic programming table of size KN, and computing each entry takes $O(N)$ time, resulting in an $O(N^2K)$-time algorithm. Choi and Narahari [28], and Manne and Olstad [79] reduced the complexity of this scheme to $O(NK)$ and $O((N-K)K)$, respectively. Pınar and Aykanat [93] presented enhancements to limit the search space of each separator by exploiting upper and lower bounds on the optimal solution value for better practical performance.

3.2.3 Parametric Search

Parametric search algorithms rely on two components: a probing operation to determine if a solution exists whose bottleneck value is no greater than a specified value, and a method to search the space of candidate values. The probe algorithm can be computed in only $O(K \log N)$ time by using binary search on the prefix-summed workload array. Below, we summarize algorithms to search the space of bottleneck values.

3.2.3.1 Nicol's Algorithm

Nicol's algorithm [84] exploits the fact that any candidate B value is equal to the weight of a task subchain. A naive solution is to generate all subchain weights, sort them, and then use binary search to find the minimum value for which a probe succeeds. Nicol's algorithm efficiently searches for this subchain by considering each processor in order as a candidate bottleneck processor. For each processor $\mathcal{P}_p$, the algorithm does a binary search for the smallest index that will make $\mathcal{P}_p$ the bottleneck processor. With the $O(K \log N)$ cost of each probing, Nicol's algorithm runs in $O(N + (K \log N)^2)$ time.

Pınar and Aykanat [93] improved Nicol's algorithm by utilizing the following simple facts. If the probe function succeeds (fails) for some B, then probe function will succeed (fail) for any $B' \geq (\leq)\ B$. Therefore by keeping the smallest B that succeeded and the largest B that failed, unnecessary probing is eliminated, which drastically improves runtime performance [93].

3.2.3.2 Bidding Algorithm

The bidding algorithm [92, 93] starts with a lower bound and proceeds by gradually increasing this bound until a feasible solution value is reached. The increments are chosen to be minimal so that the first feasible bottleneck value is optimal. Consider the partition generated by a failed probe call that loads the first $K-1$ processors maximally not to exceed the specified probe value. To find the next bottleneck value, processors bid with the bottleneck value that would add one more task to their domain, and the minimum bid among the processors is chosen to be the next bottleneck value. The bidding algorithm moves each one of the K separators for $O(N)$ positions in the worst case, where choosing the new bottleneck value takes $O(\log K)$ time using a priority queue. This makes the complexity of the algorithm $O(NK \log K)$.

3.2.3.3 Bisection Algorithms

The bisection algorithm starts with a lower and an upper bound on the solution value and uses binary search in this interval. If the solution value is known to be an integer, then the bisection algorithm finds an optimal solution. Otherwise, it is an ϵ-approximation algorithm, where ϵ is the user defined accuracy for the solution. The bisection algorithm requires $O(\log(w_{\max}/\epsilon))$ probe calls, with $O(N + K \log N \log(w_{\max}/\epsilon))$ overall complexity.

Pınar and Aykanat [93] enhanced the bisection algorithm by updating the lower and upper bounds to realizable bottleneck values (subchain weights). After a successful probe, the upper bound can be set to be the bottleneck value of the partition generated by the probe function, and after a failed probe, the lower bound can be set to be the smallest value that might succeed, as in the bidding algorithm. These enhancements transform the bisection algorithm to an exact algorithm, as opposed to an ϵ-approximation algorithm.

3.3 Proposed CCP Algorithms for Heterogeneous Systems

The algorithms we propose in this section extend the techniques for homogenous CCP to heterogeneous CCP. All algorithms discussed in this section require an initial prefix-sum operation on the task-weight array $\mathcal{W}$ for the efficiency of subsequent subchain-weight computations. The prefix-sum operation replaces the ith entry $\mathcal{W}[i]$ with the sum of the first i entries ($\sum_{h=1}^{i} w_h$) so that computational weight W_{ij} of a task subchain $\mathcal{T}_{ij}$ can be efficiently determined as $\mathcal{W}[j]-\mathcal{W}[i-1]$ in $O(1)$ time. In our discussions, $\mathcal{W}$ is used to refer to the prefix-summed $\mathcal{W}$ array, and $O(N)$ cost of this initial prefix-sum operation is considered in the complexity analysis. Similarly, $E_{a,b}$ can be computed in $O(1)$ time on a prefix-summed processor-speed array. In all algorithms, we focus only on finding the optimal solution value, since an optimal solution can be easily constructed, once the optimal solution value is known.

Unless otherwise stated, `BINSEARCH` represents a binary search that finds the index to the element that is closest to the target value. There are variants of `BINSEARCH` to find the index of the greatest element not greater than the target value, and we will state whenever such variants are needed. `BINSEARCH` takes four parameters: the array to search, the start and end indices of the sub-array, and the target value. The range parameters are optional, and their absence means that the search will be performed on the whole array.

3.3.1 Heuristics

We propose a heuristic, `RB`, based on the recursive bisection idea. During each bisection, `RB` performs a two step process. First, it divides the current processor chain $\mathcal{P}_{p,r}$ into two subchains $\mathcal{P}_{p,q}$ and $\mathcal{P}_{q+1,r}$. Then, it divides the current task chain $\mathcal{T}_{h,j}$ into two subchains $\mathcal{T}_{h,i}$ and $\mathcal{T}_{i+1,j}$ in proportion to the computational powers of the respective processor subchains. That is, the task separator index i is chosen such that the ratio $W_{h,i}/W_{i+1,j}$ is as close to the ratio $E_{p,q}/E_{q+1,r}$ as possible. RB achieves optimal bisections at each level; however, the quality of the overall partition may be far away from that of the optimal solution.

We have investigated two metrics for bisecting the processor chain: chain length and chain processing power. The chain length metric divides the current processor chain $\mathcal{P}_{p,r}$ into two equal-length processor subchains, whereas the chain processing power metric divides $\mathcal{P}_{p,r}$ into two equal-power subchains. Since the first metric performed slightly better than the second one in our experiments, we will only discuss the chain length metric here. The pseudocode of the RB algorithm is given in Algorithm 3.1, where the initial invocation takes its parameters as $(\mathcal{W}, \mathcal{E}, 1, K)$ with $s_0 = 0$ and $s_K = N$. Note that s_{p-1} and s_r are already determined at higher levels of recursion. *Wtot* is the total weight of current task subchain, and *Wfirst* is the weight for the first processor subchain in proportion to its processing speed. We need to add $W_{1,s_{p-1}}$ to *Wfirst* to seek s_q in the prefix-summed $\mathcal{W}$ array.

We also propose a generalization of Miguet and Pierson's heuristic, `MP` [81].

Algorithm 3.1 RB($\mathcal{W}$, $\mathcal{E}$, p, r)

1: **if** $p = r$ **then**
2: **return**
3: $Wtot \leftarrow W_{s_{p-1}+1,s_r}$
4: $q \leftarrow (p + r - 1)/2$
5: $Wfirst \leftarrow Wtot \times E_{p,q}/E_{p,r}$
6: $W \leftarrow Wfirst + W_{1,s_{p-1}}$
7: $s_q \leftarrow$ BinSearch($\mathcal{W}, s_{p-1}, s_r, W$)
8: RB($\mathcal{W}, \mathcal{E}, p, q$)
9: RB($\mathcal{W}, \mathcal{E}, q + 1, r$)

Algorithm 3.2 MP($\mathcal{W}$, N, $\mathcal{E}$, P)

1: **for** $p \leftarrow 1$ **to** P **do**
2: $w \leftarrow W_{1,N} \times E_{1,p}/E_{1,P}$
3: $s_p \leftarrow$ BinSearch($\mathcal{W}, s_{p-1}, N, w$)

`MP` computes the separator index of each processor by considering that processor as a division point for the whole processor chain. In our version, the load assigned to the processor chain $\mathcal{P}_{1,p}$ is set to be proportional to the computational power $E_{1,p}$ of this subchain, as shown in Algorithm 3.2.

Both `RB` and `MP` can be implemented in $O(N + K \log N)$ time, where the $O(N)$ time is due to the initial prefix-sum operation on the task-weight array.

Below, we investigate the theoretical bounds on the quality of these two heuristics. We assume K is a power of 2 for simplicity.

Lemma 3.3.1 *B_{RB} is upper bounded by $B^* + w_{\max}/e_{\min} - w_{\max}/(Ke_{\min})$.*

Proof: We use induction, and the basis is easy to show for $K = 2$. For the inductive step, assume the hypothesis holds for any number of processors less than K. Consider the first bisection, where the processors are split into two subchains, each containing $K/2$ processors. Let the total processing power in the left subchain be E_{left}. `RB` will distribute the workload array between the left and right processor subchains as evenly as possible. There will be a task t_i such that the left processor subchain will weigh more than the right subchain if t_i is assigned to the left subchain, and vice versa. Without loss of generality, assume

that t_i is assigned to the left subchain. In the worst case, t_i is the maximum weighted task, and the total task weight assigned to the left subchain, W_{left}, can be upper bounded by

$$W_{\text{left}} \leq \frac{(W_{\text{tot}} + w_{\max})E_{\text{left}}}{E_{\text{tot}}}. \tag{3.4}$$

Using the inductive hypothesis, the bottleneck value among the processors of the left processor subchain can be upper bounded as follows.

$$\begin{aligned}
B_{RB} &\leq \frac{W_{\text{left}}}{E_{\text{left}}} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}K/2} && (3.5)\\
&\leq \frac{W_{\text{tot}} + w_{\max}}{E_{\text{tot}}} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}K/2} && (3.6)\\
&= B^* + \frac{w_{\max}}{E_{\text{tot}}} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}K/2} && (3.7)\\
&\leq B^* + \frac{w_{\max}}{e_{\min}K} + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{e_{\min}K/2} && (3.8)\\
&= B^* + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{Ke_{\min}} && (3.9)
\end{aligned}$$

The same bound applies to the right processor subchain directly by the inductive hypothesis, since right processor subchain is already underloaded. ■

Lemma 3.3.2 *B_{MP} is upper bounded by $B^* + w_{\max}/e_{\min}$.*

Proof: Let the sequence $\langle s_0, s_1, \ldots, s_K \rangle$ be the partition constructed by `MP`. For a processor $\mathcal{P}_p$, s_p is chosen to be the separator that best divides $\mathcal{P}_{1,p}$ and $\mathcal{P}_{p+1,K}$. Based on our discussion of bipartitioning quality in the proof of Lemma 3.3.1, W_{1,s_p} is bounded by

$$E_{1,p}B^* - \frac{w_{\max}}{2} \leq W_{1,s_p} \leq E_{1,p}B^* + \frac{w_{\max}}{2}$$

So, the load of processor p is upper bounded by

$$\begin{aligned}
\frac{W_{1,s_p} - W_{1,s_{p-1}}}{e_p} &\leq \frac{E_{1,p}B^* + w_{\max}/2 - E_{1,p-1}B^* + w_{\max}/2}{e_p} && (3.10)\\
&= B^* + \frac{w_{\max}}{e_p} && (3.11)\\
&\leq B^* + \frac{w_{\max}}{e_{\min}} && (3.12)
\end{aligned}$$

The bottleneck value of a partition constructed by MP cannot be greater than $B^* + w_{\max}/e_{\min}$. ■

3.3.2 Dynamic Programming

The overlapping subproblems and the optimal substructure properties of the homogenous CCP can be extended to the heterogeneous CCP, and thus enabling dynamic programming solutions. The recursive definition for the bottleneck value of an optimal partition can be derived as

$$B_i^p = \min_{0 \le j \le i} \left\{ \max \left\{ B_j^{p-1}, \frac{W_{j+1,i}}{e_p} \right\} \right\} \tag{3.13}$$

for the heterogeneous case. As in the homogenous case, B_i^p denotes the optimal solution value for partitioning the first i tasks onto the first p processors. This definition results in an $O(N^2K)$-time DP algorithm.

We generalize the observations of Choi and Narahari [28] to develop an $O(NK)$-time algorithm for heterogeneous systems as follows. Their first observation relies on the fact that the optimal position of the separator for partitioning the first i tasks cannot be to the left of the optimal position for the first $i-1$ tasks, i.e., $j_i^p \ge j_{i-1}^p$. Their second observation is that we need to advance a separator index only when the last part is overloaded and can stop when this is no longer the case, i.e., $B_j^{p-1} \ge W_{j+1,i}/e_p$. Then an optimal j_i^p can be chosen to correspond to the minimum of $\max\{B_j^{p-1}, W_{j+1,i}/e_p\}$ and $\max\{B_{j-1}^{p-1}, W_{j,i}/e_p\}$. That is, the recursive definition becomes:

$$B_i^p = \max \left\{ B_{j_i^p}^{p-1}, \frac{W_{j_i^p+1,i}}{e_p} \right\}, \tag{3.14}$$

$$\text{where } j_i^p = \text{argmin}_{j_{i-1}^p \le j \le i} \left\{ \max \left\{ B_j^{p-1}, \frac{W_{j+1,i}}{e_p} \right\} \right\}. \tag{3.15}$$

It is clear that the search ranges of separators overlap at only one position, and thus we can compute all B_i^p entries for $1 \le i \le N$ in only one pass over the task subchain. This reduces the complexity of the algorithm to $O(NK)$. Algorithm 3.3 presents this algorithm.

Algorithm 3.3 DP($\mathcal{W}$, N, P, $\mathcal{E}$)

```
1: for p ← 1 to P do
2:     B[p, 0] ← 0
3: for i ← 1 to N do
4:     B[1, i] ← W_{1,i}/e_1
5: for p ← 2 to P do
6:     j ← 0
7:     for i ← 1 to N do
8:         if W_{j+1,i}/e_p ≤ B[p − 1, j] then
9:             B[p, i] ← B[p − 1, j]
10:        else
11:            repeat
12:                j ← j + 1
13:            until W_{j+1,i}/e_p ≤ B[p − 1, j] or j ≥ i
14:            if W_{j,i}/e_p < B[p − 1, j] then
15:                B[p, i] ← W_{j,i}/e_p
16:                j ← j − 1
17:            else
18:                B[p, i] ← B[p − 1, j]
19: return B_opt ← B[P, N]
```

In the homogenous case, Manne and Olstad [79] reduced the complexity further to $O((N-K)K)$ by observing that there is no merit in leaving a processor empty, and thus the search for j_i^p can start at p instead of 1. However, this does not apply to the heterogeneous CCP, since it might be beneficial to leave a processor empty.

Alternatively, we propose another DP algorithm by extending the `DP+` algorithm (`DP` algorithm with static separator-index bounding) of Pınar and Aykanat [93] for the heterogeneous case. `DP+` limits the search space of each separator to avoid redundant calculation of B_i^p values. `DP+` achieves this separator index bounding by running left-to-right and right-to-left probe functions with the upper and lower bounds on the optimal bottleneck value.

We extend the probing operation to the heterogeneous case as shown in Algorithms 3.5 and 3.6. In the figure, `LR-PROBE` and `RL-PROBE` denote the left-to-right probe and right-to-left probe, respectively. These algorithms not only decide

Algorithm 3.4 DP+($\mathcal{W}$, N, $\mathcal{E}$, P, SL, SH)

1: **for** $p \leftarrow 1$ **to** P **do**
2: $\quad B[p, 0] \leftarrow 0$
3: **for** $i \leftarrow SL_1$ **to** SH_1 **do**
4: $\quad B[1, i] \leftarrow W_{1,i}/e_1$
5: **for** $p \leftarrow 2$ **to** P **do**
6: $\quad j \leftarrow SL_{p-1}$
7: $\quad$ **for** $i \leftarrow SL_p$ **to** SH_p **do**
8: $\quad\quad$ **if** $W_{j+1,i}/e_p \leq B[p-1, j]$ **then**
9: $\quad\quad\quad B[p, i] \leftarrow B[p-1, j]$
10: $\quad\quad$ **else**
11: $\quad\quad\quad$ **repeat**
12: $\quad\quad\quad\quad j \leftarrow j + 1$
13: $\quad\quad\quad$ **until** $W_{j+1,i}/e_p \leq B[p-1, j]$ **or** $j \geq i$
14: $\quad\quad\quad$ **if** $W_{j,i}/e_p < B[p-1, j]$ **then**
15: $\quad\quad\quad\quad B[p, i] \leftarrow W_{j,i}/e_p$
16: $\quad\quad\quad\quad j \leftarrow j - 1$
17: $\quad\quad\quad$ **else**
18: $\quad\quad\quad\quad B[p, i] \leftarrow B[p-1, j]$
19: **return** $B_{\text{opt}} \leftarrow B[P, N]$

whether a candidate value is a feasible bottleneck value, but they also set the separator index (s_p) values for their greedy approach. In `LR-PROBE`, `BINSEARCH`($\mathcal{W}, w$) refers to a binary search algorithm that searches $\mathcal{W}$ for the largest index m such that $W_{1,m} \leq w$. Similarly, in `RL-PROBE`, `BINSEARCH`($\mathcal{W}, w$) searches $\mathcal{W}$ for the smallest index m such that $W_{1,m} \geq w$.

`DP+`, as presented in Algorithm 3.4, uses Lemma 3.3.3 to limit the search space of s_p values.

Lemma 3.3.3 *For a given heterogeneous CCP instance* $(\mathcal{W}, N, \mathcal{E}, K)$*, a feasible bottleneck value UB and a lower bound on the bottleneck value LB; let the sequences* $\Pi^1 = \langle h^1_0, h^1_1, \ldots, h^1_K \rangle$*,* $\Pi^2 = \langle l^2_0, l^2_1, \ldots, l^2_K \rangle$*,* $\Pi^3 = \langle l^3_0, l^3_1, \ldots, l^3_K \rangle$ *and* $\Pi^4 = \langle h^4_0, h^4_1, \ldots, h^4_K \rangle$ *be the partitions constructed by* `LR-PROBE`*(UB),* `RL-PROBE`*(UB),* `LR-PROBE`*(LB) and* `RL-PROBE`*(LB), respectively. Then, an optimal partition* $\Pi_{\text{opt}} = \langle s_0, s_1, \ldots, s_K \rangle$ *satisfies* $SL_p \leq s_p \leq SH_p$ *for all* $1 \leq p \leq K$*, where* $SL_p = \max\{l^2_p, l^3_p\}$ *and* $SH_p = \min\{h^1_p, h^4_p\}$*.*

Algorithm 3.5 LR-PROBE($\mathcal{W}$, N, $\mathcal{E}$, P, B)

1: $sum \leftarrow 0$
2: **for** $p \leftarrow 1$ **to** $P-1$ **do**
3: $\quad myB \leftarrow B \times e_p$
4: $\quad Bsum \leftarrow sum + myB$
5: $\quad m \leftarrow$ BINSEARCH($\mathcal{W}$, $Bsum$)
6: $\quad sum \leftarrow \mathcal{W}_{1,m}$
7: $\quad s_p \leftarrow m$
8: **if** $sum + B \times e_P \geq W_{1,N}$ **then**
9: $\quad$ **return** TRUE
10: **else**
11: $\quad$ **return** FALSE

Algorithm 3.6 RL-PROBE($\mathcal{W}$, N, $\mathcal{E}$, P, B)

1: $sum \leftarrow W_{1,N}$
2: **for** $p \leftarrow P$ **downto** 2 **do**
3: $\quad myB \leftarrow B \times e_p$
4: $\quad Bsum \leftarrow sum - myB$
5: $\quad m \leftarrow$ BINSEARCH($\mathcal{W}$, $Bsum$)
6: $\quad sum \leftarrow \mathcal{W}_{1,m}$
7: $\quad s_{p-1} \leftarrow m$
8: **if** $sum - B \times e_1 \leq 0$ **then**
9: $\quad$ **return** TRUE
10: **else**
11: $\quad$ **return** FALSE

Proof: We know that any feasible bottleneck value is greater than or equal to the optimal bottleneck value, i.e., $UB \geq B_{\text{opt}}$. Consider h_p^1, which is the largest index such that the first h_p^1 tasks can be partitioned over p processors without exceeding UB. Then $s_p > h_p^1$ implies $B_{\text{opt}} > UB$, which is a contradiction. So, $s_p \leq h_p^1$. Since, `RL-PROBE` is just the symmetric algorithm of `LR-PROBE`, the same argument proves $s_p \geq l_p^2$.

Consider the optimal partition constructed by `RL-PROBE`(B_{opt}). Since $B_{\text{opt}} \geq LB$, by the greedy property of `RL-PROBE`, $s_p \leq h_p^4$. Assume $s_p < l_p^3$ for some p, then another partition obtained by advancing the s_p value to l_p^3 does not increase the bottleneck value, since the first l_p^3 tasks are successfully partitioned over the first p processors without exceeding LB and thus B_{opt}. An optimal partition $\Pi_{\text{opt}} = \langle s_0, s_1, \ldots, s_K \rangle$ satisfies $l_p^3 \leq s_p \leq h_p^4$. ■

The lower bound LB can be initialized to the optimal lower bound when all processors are equally loaded as

$$LB = B^* = \frac{W_{\text{tot}}}{E_{\text{tot}}}. \tag{3.16}$$

An upper bound UB can be computed in practice with a fast and effective heuristic, and Lemma 3.3.1 provides a theoretically robust bound as

$$UB = B^* + \frac{w_{\max}}{e_{\min}} - \frac{w_{\max}}{Ke_{\min}}. \tag{3.17}$$

3.3.3 Parametric Search

Parametric search algorithms can be constructed with a `PROBE` function (either `LR-PROBE` given in Algorithm 3.5 or `RL-PROBE` given in Algorithm 3.6), and a method to search the space of candidate values. Below, we describe several algorithms to search the space of bottleneck values for the heterogeneous case.

3.3.3.1 Nicol's Algorithm

We revise Nicol's algorithms for heterogeneous systems as follows. The candidate B values become task subchain weights divided by processor subchain speeds. The algorithm starts with searching for the smallest j so that probing with $W_{1,j}/e_1$ succeeds, and probing with $W_{1,j-1}/e_1$ fails. This means $W_{1,j-1}/e_1 < B_{\text{opt}} \leq W_{1,j}/e_1$, and thus in an optimal solution the probe function will assign the first j tasks to the first processor if it is the bottleneck processor, and the first $j-1$ tasks to the first processor if not. Then the optimal solution value is the minimum of $W_{1,j}/e_1$ and the optimal solution value for partitioning the remaining task subchain $\mathcal{T}_{j,N}$ to the processor subchain $\mathcal{P}_{2,K}$, since any solution with a bottleneck value less than $W_{1,j}/e_1$ will assign only the first $j-1$ tasks to the first processor. Finding the j value requires $\lg N$ probes, and we repeat this search operation for all processors in order. This version of Nicol's algorithm runs in $O(N+(K \log N)^2)$ time. Algorithm 3.7 displays this algorithm.

Algorithm 3.7 NICOL($\mathcal{W}$, $\mathcal{E}$, N, P)

```
1: i_0 ← 1
2: for b ← 1 to P − 1 do
3:     ilow ← i_{b−1}
4:     ihigh ← N
5:     while ilow < ihigh do
6:         imid ← (ilow + ihigh)/2
7:         B ← W_{i_{b−1},imid}/e_b
8:         if PROBE(B) then
9:             ihigh ← imid
10:        else
11:            ilow ← imid + 1
12:    i_b ← ihigh
13:    B_b ← W_{i_{b−1},i_b}/e_b
14: B_P ← W_{i_{P−1},N}/e_P
15: return B_opt ← min_{1≤b≤P}{B_p}
```

3.3.3.2 Nicol's Algorithm with Dynamic Bottleneck-Value Bounding

By keeping the largest B that succeeded and the smallest B that failed, we can improve Nicol's algorithm by eliminating unnecessary probing. Let LB and UB represent the lower bound and upper bound for B_{opt}, respectively. If a processor cannot update LB or UB, that processor does not make any `PROBE` calls. This algorithm, presented in Algorithm 3.8, is referred to as `NICOL+`.

In the worst case, a processor makes $O(\log N)$ `PROBE` calls. But, as we will prove below, the number of probes performed by `NICOL+` cannot exceed $K \lg(1 + w_{\max}/(K e_{\min} w_{\min}))$. This analysis also improves known complexities of homogeneous version of the algorithm. Lemma 3.3.4 describes an upper bound on the number of probes performed by `NICOL+` algorithm.

Lemma 3.3.4 *The number of probes required by* `NICOL+` *is upper bounded by* $K \lg(1 + (UB - LB) / (K w_{\min}))$.

Proof: Consider the first step of the algorithm, where we search for the smallest separator index that makes the first processor the bottleneck processor. We can restrict this search in a range that covers only those indices for which the weight

Algorithm 3.8 NICOL+($\mathcal{W}$, $\mathcal{E}$, N, P)

```
1: i_0 ← 1
2: LB ← B* ← W_{1,N}/E_{1,P}
3: UB ← LB + w_max × (1/e_min − 1/E_tot)
4: for b ← 1 to P − 1 do
5:     ilow ← i_{b−1}
6:     ihigh ← N
7:     while ilow < ihigh do
8:         imid ← (ilow + ihigh)/2
9:         B ← W_{i_{b−1},imid}/e_b
10:        if LB ≤ B < UB then
11:            if PROBE(B) then
12:                ihigh ← imid
13:                UB ← B
14:            else
15:                ilow ← imid + 1
16:                LB ← B
17:        else if B ≥ UB then
18:            ihigh ← imid
19:        else
20:            ilow ← imid + 1
21:    i_b ← ihigh
22:    B_b ← W_{i_{b−1},i_b}/e_b
23: B_P ← W_{i_{P−1},N}/e_P
24: return B_opt ← min_{1≤b≤P}{B_p}
```

of the first chain will be in the $[LB, UB]$ interval. If there are n_1 tasks in this range, NICOL+ will require $\lg n_1$ probes. This means that the $[LB, UB]$ interval is narrowed by at least $(n_1 - 1)w_{\min}$ after the first step.

Let k_p be the number of probes by the pth processor. Since k_p probes narrow the $[LB, UB]$ interval by $\left(2^{k_p} - 1\right) w_{\min}$, we have

$$\left(\left(2^{k_1} - 1\right) + \left(2^{k_2} - 1\right) + \ldots + \left(2^{k_{K-1}} - 1\right)\right) w_{min} \leq UB - LB,$$

and thus $2^{k_1} + 2^{k_2} + \ldots + 2^{k_{K-1}} \leq \dfrac{UB - LB}{w_{\min}} + K - 1$. The corresponding total number of probes is $\sum_{p=1}^{K-1} k_p$, which reaches its maximum when $\sum_{p=1}^{K-1} 2^{k_p}$ is maximum and $k_1 = k_2 = \ldots = k_{K-1} = k$ for some k. In that case,

$$(K-1)2^k \leq \frac{UB - LB}{w_{\min}} + K - 1$$

and thus

$$k \leq \lg\left(1 + \frac{UB - LB}{w_{\min}(K-1)}\right).$$

So, the total number of probes performed by `NICOL+` is upper bounded by:

$$\sum_{p=1}^{K-1} k_p \leq (K-1)k \leq (K-1)\lg\left(1 + \frac{UB - LB}{w_{\min}(K-1)}\right) < K\lg\left(1 + \frac{UB - LB}{w_{\min}K}\right) \tag{3.18}$$

■

Corollary 3.3.1 `NICOL+` *requires at most* $K\lg(1 + w_{\max}/(Ke_{\min}w_{\min}))$ *probes for heterogeneous, and* $K\lg(1 + w_{\max}/(Kw_{\min}))$ *probes for homogeneous systems.*

`NICOL+` runs in $O(N + K^2 \log N \log(1 + w_{\max}/(Ke_{\min}w_{\min})))$ time, with the $O(K \log N)$ cost of a `PROBE` call. In most configurations, $w_{\max}/(e_{\min}w_{\min}K)$ is very small, and is $O(1)$ if $Ke_{\min} = \Omega(w_{\max}/w_{\min})$. In that case, the runtime complexity of `NICOL+` reduces to $O(N + K^2 \log N)$.

We also studied expected-case complexity analysis of `NICOL+` algorithm. Our analysis further indicate that the expected complexity of `NICOL+` is better than the worst-case complexities. The experiments also validate the theoretical findings. We present the expected complexity of `NICOL+` in Appendix A.1.

3.3.3.3 Bidding Algorithm

For heterogeneous systems, the bidding algorithm uses the lower bound given in Eq. 3.16 for optimal bottleneck value, and gradually increases this lower bound. The bid of each processor P_p, for $p = 1, 2, \ldots, K-1$, is calculated as $W_{s_{p-1}+1, s_p+1} / e_p$, which is equal to the load of P_p if it also executes the first task of PK_{p+1} in addition to its current load. Then, the algorithm selects the processor with the minimum bid value so that this bid value becomes the next bottleneck value to be considered for feasibility. The processors following the bottleneck processor in the processor chain are processed in order, except the last processor. The separator indices of these processors are adjusted accordingly

Algorithm 3.9 BIDDING($\mathcal{W}$, N, $\mathcal{E}$, P)

1: $minBid \leftarrow W_{1,N}/E_{1,P}$
2: LR-PROBE($\mathcal{W}, N, \mathcal{E}, P, minBid$)
3: **for** $p \leftarrow 1$ **to** $P-1$ **do**
4: $\quad bids[p] \leftarrow W_{s_{p-1}+1,s_p+1}/e_p$
5: $Q \leftarrow$ BUILD-HEAP(P)
6: **repeat**
7: $\quad minP \leftarrow$ EXTRACT-MIN(Q)
8: $\quad wlast \leftarrow W_{s_{P-1}+1,N}/e_P$
9: $\quad minBid \leftarrow bids[minP]$
10: $\quad$ **if** $minBid < wlast$ **then**
11: $\quad\quad$ **for** $p \leftarrow minP$ **to** $P-1$ **do**
12: $\quad\quad\quad s_p \leftarrow$ BINSEARCH($\mathcal{W}, minBid \times e_p + W_{1,s_{p-1}}$)
13: $\quad\quad\quad previousBid \leftarrow bids[p]$
14: $\quad\quad\quad bids[p] \leftarrow W_{s_{p-1}+1,s_p}/e_p$
15: $\quad\quad\quad$ **if** $bids[p] > previousBid$ **then**
16: $\quad\quad\quad\quad$ INCREASE-KEY(Q, p)
17: $\quad\quad\quad$ **else if** $bids[p] < previousBid$ **then**
18: $\quad\quad\quad\quad$ DECREASE-KEY(Q, p)
19: **until** $minBid \geq wlast$

so that the processors are maximally loaded not to exceed that new bottleneck value. The load of the last processor determines the feasibility of the current bottleneck value. If current bottleneck value is not feasible, the process repeats. Algorithm 3.9 presents the bidding algorithm, which uses a min-priority queue that maintains the processors keyed according to their bid values. In the figure, `BUILD-HEAP`, `EXTRACT-MIN`, `INCREASE-KEY` and `DECREASE-KEY` functions refer to the respective priority queue operations [34].

In the worst case, the bidding algorithm moves K separators for $O(N)$ positions. Choosing a new bottleneck value takes $O(\log K)$ time using a binary heap implementation of the priority queue. Totally the complexity of the algorithm is $O(NK \log K)$ in the worst case. Despite this high worst-case complexity, the bidding algorithm is quite fast in practice.

Algorithm 3.10 BISECTION($\mathcal{W}$, N, $\mathcal{E}$, P, ϵ)

1: $LB \leftarrow W_{1,N}/E_{1,P}$
2: $UB \leftarrow LB + w_{\max}/e_{\min}$
3: **while** $UB - LB \geq \epsilon$ **do**
4: $\quad midB \leftarrow (UB + LB)/2$
5: $\quad$ **if** PROBE($midB$) **then**
6: $\quad\quad UB \leftarrow midB$
7: $\quad$ **else**
8: $\quad\quad LB \leftarrow midB$
9: **return** UB

3.3.3.4 Bisection Algorithm

For heterogeneous systems, the bisection algorithm can use the LB and UB values given in Eqs. 3.16 and 3.17. A binary search on this $[LB, UB]$ interval requires $O(\log(w_{\max}/(\epsilon E_{\text{tot}})))$ probes, thus leading to an $O(\log(w_{\max}/(\epsilon E_{\text{tot}}))K \log N)$-time algorithm, where ϵ is the specified accuracy of the algorithm. Algorithm 3.10 presents this ϵ-approximation bisection algorithm. We should note that, although the homogenous version of this algorithm becomes an exact algorithm for integer-valued workload arrays by setting $\epsilon = 1$, this is not the case for heterogeneous systems.

We enhance this bisection algorithm to be an exact algorithm for heterogeneous systems by extending the scheme proposed by Pınar and Aykanat [93] for homogenous systems. After each probe, we move lower and upper bounds to realizable bottleneck values, as opposed to the probed value. In heterogeneous systems, realizable bottleneck values are subchain weights divided by appropriate processor speeds. After a successful probe, we decrease UB to the bottleneck value of the partition constructed by the probe, and after a failed probe we increase LB to the bid value as described for the bidding algorithm in Section 3.3.3.3. Each probe eliminates at least one candidate bottleneck value, and thus the bisection algorithm terminates in a finite number of steps with an optimal solution. Algorithm 3.11 displays the exact bisection algorithm.

Algorithm 3.11 EXACT-BISECTION($\mathcal{W}$, N, $\mathcal{E}$, P)

```
1: LB ← W_{1,N}/E_{1,P}
2: UB ← LB + w_max/e_min
3: while UB > LB do
4:     midB ← (UB + LB)/2
5:     if LR-PROBE(midB ) then
6:         UB ← min_{1≤p≤P} W_{s_{p-1}+1,s_p}/e_p
7:     else
8:         LB ← min_{1≤p≤P-1} W_{s_{p-1}+1,s_p+1}/e_p
9: return UB
```

3.4 Chain Partitioning (CP) Problem for Heterogeneous Systems

In this section, we study the problem of partitioning a chain of tasks onto a set of processors, as opposed to a chain of processors. The solution to this problem is not only separators on the task chain, but also processor-to-subchain assignments. Thus, we define a mapping $\mathcal{M}$ as a partition $\Pi = \langle s_0 = 0, s_1, \ldots, s_K = N \rangle$ of the given task chain $\mathcal{T} = \langle t_1, t_2, \ldots t_N \rangle$ with $s_p \leq s_{p+1}$ for $0 \leq p < K$, and a permutation $\langle \pi_1, \pi_2, \ldots, \pi_K \rangle$ of the given set of K processors $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$. According to this mapping, the pth task subchain $\langle t_{s_{p-1}+1}, \ldots, t_{s_p} \rangle$ is executed on processor P_{π_p}. The cost $C(\mathcal{M})$ of a mapping $\mathcal{M}$ is the maximum subchain computation time, determined by the subchain weight and the execution speed of the assigned processor, i.e.,

$$C(\mathcal{M}) = \max_{1 \leq p \leq K} \left\{ \frac{W_{s_{p-1}+1,s_p}}{e_{\pi_p}} \right\}.$$

We will prove that the CP problem is NP-complete. The decision problem for the CP problem for heterogeneous systems is as follows.

Given a chain of tasks $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$, a weight $w_i \in Z^+$ for each $t_i \in \mathcal{T}$, a set of processors $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ with $K < N$, an execution speed $e_p \in Z^+$ for each $P_p \in \mathcal{P}$, and a bound B, decide if there exists a mapping $\mathcal{M}$ of $\mathcal{T}$ onto $\mathcal{P}$ such that $C(\mathcal{M}) \leq B$.

Theorem 3.4.1 *The CP problem for heterogeneous systems is NP-complete.*

Proof: We use reduction from the 3-Partition (3P) problem. A pseudo-polynomial transformation suffices, because 3P problem is NP-complete in the strong sense (i.e., there is no pseudo-polynomial time algorithm for the problem unless P=NP). The 3P problem is stated in [50] as follows.

Given a finite set $\mathcal{A}$ *of* $3m$ *elements, a bound* $B \in Z^+$*, and a cost* $c_i \in Z^+$ *for each* $a_i \in \mathcal{A}$*, where* $\sum_{a_i \in \mathcal{A}} c_i = mB$ *and each* c_i *satisfies* $B/4 < c_i < B/2$*, decide if* $\mathcal{A}$ *can be partitioned into* m *disjoint sets* $S_1, S_2, \ldots, S_m$ *such that* $\sum_{a_i \in S_p} c_i = B$ *for* $p = 1, 2, \ldots, m$.

For a given instance of the 3P problem, the corresponding CP problem is constructed as follows.

- The number of tasks N is $m(B+1)-1$. The weight of every $(B+1)$st task is B, (i.e., $w_i = B$ for $i \text{ } mod \text{ } (B+1) = 0$), and the weights of all other tasks are 1.
- The number of processors K is $4m-1$. The first $m-1$ processors have execution speeds of B, (i.e., $e_p = B$ for $p = 1, 2, \ldots, m-1$), and the remaining processors have execution speeds equal to the costs of items in the 3P problem (i.e., $e_p = c_{p-m+1}$ for $p = m, \ldots, 4m-1$).

We claim that there is a solution to the 3P problem if and only if there is a mapping $\mathcal{M}$ with cost $C(\mathcal{M}) = 1$ for the CP problem. The following observations constitute the basis for our proof.

- The processors with execution speeds of B must be mapped to tasks with weight B to have a solution with cost $C(\mathcal{M}) = 1$, because the execution speeds of all other processors are $\leq B/2$. These processors (tasks) serve as divider processors (tasks).
- The total weight of the chain is $3m + (m-1)B = (B+3)m - B$. The sum of execution speeds of all processors is also $(m-1)B + 3m = (B+3)m - B$. This forces each processor to be assigned a load with value equal to its execution speed to achieve a mapping with cost $C(\mathcal{M}) = 1$.

As noted above, the divider processors should be assigned to the divider tasks. Between two successive divider tasks there is a subchain of B unit-weight tasks with total weight B, which must be assigned to a subset of processors with total execution speed B. Since there are m such subchains, the same grouping of the processors is also valid for grouping c_i values in the 3P problem. Thus the 3P problem can be reduced to the CP problem, proving the CP problem is NP-hard.

The cost of a given mapping can be computed in polynomial time, thus the problem is in NP. Thus we can conclude that the chain partitioning problem for heterogeneous systems is NP-Complete. ■

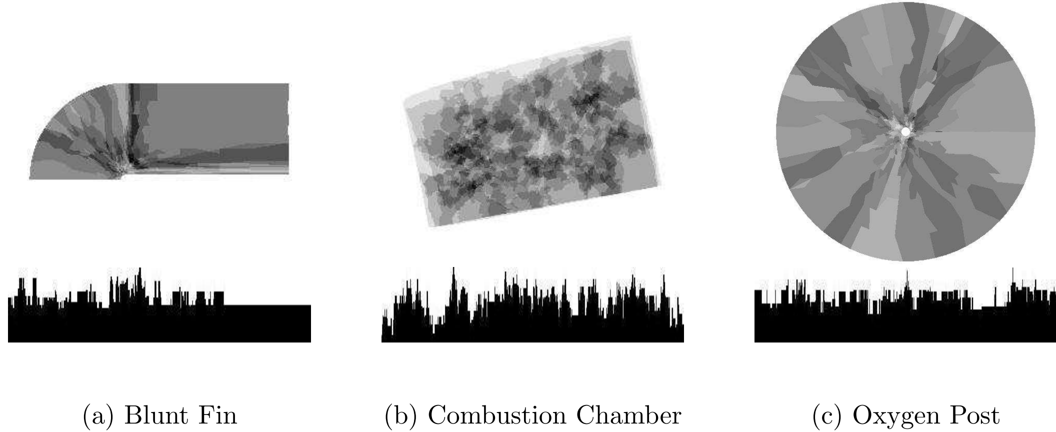
This complexity shows that we need to resort to heuristics for practical solutions to the CP problem. With the nearly perfect balance results and extremely fast runtimes as we will present in Section 3.5.2, CCP algorithms can serve as good heuristics for the CP problem. We tried this approach by finding optimal CCP solutions for randomly ordered processor chains of a CP instance. We observed that the sensitivity to processor ordering is quite low. You can find a description of these studies in Section 3.5.3. We also tried improvement techniques, where we swapped processors in the chain to decrease the bottleneck value, but the improvements were modest and could hardly compensate for the increase in runtimes.

3.5 Experimental Results

3.5.1 Experimental Setup

The 1D task arrays used in both CCP and CP experiments were derived from two different applications: image-space-parallel direct volume rendering and row-parallel sparse matrix vector multiplication.

Direct volume rendering experiments are performed on three curvilinear datasets from NASA Ames Research Center, namely *Blunt Fin* (blunt) [62], *Combustion Chamber* (comb) [38], and *Oxygen Post* (post) [96]. These datasets



(a) Blunt Fin (b) Combustion Chamber (c) Oxygen Post

Figure 3.1: Visualization of direct volume rendering dataset workloads. Top: workload distributions of 2D task arrays. Bottom: histograms showing weight distributions of 1D task chains.

are processed using the tetrahedralization techniques described in [51] and [100] to produce three-dimensional (3D) unstructured volumetric datasets. The two-dimensional (2D) workload arrays are constructed by projecting 3D volumetric datasets onto 2D screens of resolution 256×256 using the workload criteria of image-space-parallel direct volume rendering algorithm described in [14]. Here, the rendering operations associated with the individual pixels of the screen constitute the computational tasks of the application. The resulting 2D task array is then mapped to a 1D task array using Hilbert space-filling-curve traversal [91]. The workload distributions of the 2D task arrays are visualized in Fig. 3.1, where darker areas represent more weighted tasks. The histograms at the bottom of the 2D pictures show the weight distributions of the resulting 1D task arrays.

In the sparse matrix experiments, we consider rowwise block partitioning of the matrices obtained from University of Florida Sparse Matrix Collection [36]. In row-parallel matrix vector multiplies, the rows correspond to the tasks to be partitioned, and the number of nonzeros in each row is the weight of the corresponding task. The nonzero distributions of the sparse matrices are shown in Fig. 3.2. The histograms on the right sides of the visualizations represent the number of nonzeros in each row.

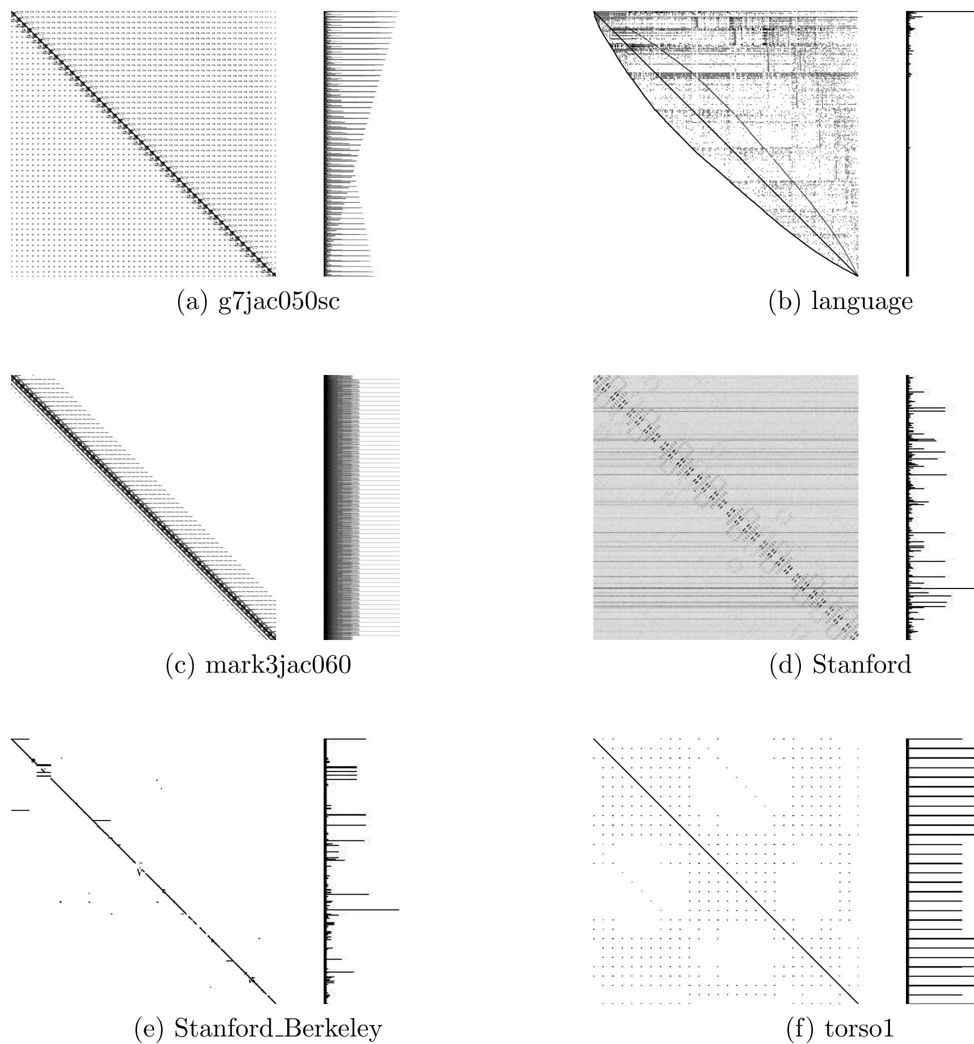

(a) g7jac050sc
(b) language
(c) mark3jac060
(d) Stanford
(e) Stanford_Berkeley
(f) torso1

Figure 3.2: Visualization of sparse matrix dataset workloads. Left: non-zero distributions of the sparse matrices. Right: histograms showing weight distributions of the 1D task chains.

Table 3.2: Properties of the test set

Name	No. of tasks N	Workload			
		Total	Per task		
		W_{tot}	w_{avg}	$w_{\min}$	$w_{\max}$
Volume rendering dataset					
`blunt`	20.6 K	1.9 M	90.95	36	171
`comb`	32.2 K	2.1 M	64.58	14	149
`post`	49.0 K	5.4 M	109.73	33	199
Sparse matrix dataset					
`g7jac050sc`	14.7 K	0.2 M	10.70	2	149
`language`	399.1 K	1.2 M	3.05	1	11555
`mark3jac060`	27.4 K	0.2 M	6.22	2	44
`Stanford`	261.6 K	2.3 M	8.84	1	38606
`Stanford_Berkeley`	615.4 K	7.6 M	12.32	1	83448
`torso1`	116.2 K	8.5 M	73.32	9	3263

Table 3.2 displays the properties of the 1D task chains used in our experiments. In the volume rendering dataset, the number of tasks is considerably less than the screen resolution, because zero-weight tasks are omitted. In the sparse matrix dataset, the number of tasks is equal to the number of rows.

In both CCP and CP experiments, $P = 32, 64, 128,\ 256, 512, 1024$, and 2048-way partitioning of the 1D task arrays were performed. We experimented with different variances of processor speeds, where the processors speeds were chosen uniformly distributed in the 1–4, 1–8, and 1–16 ranges.

In the experiments, the P-way partitioning of a given task chain for a given processor speed range constitutes a partitioning instance. As randomization is used in determining processor speeds, each task chain was partitioned onto 20 different uniformly random processor chains/sets for each speed range, and average performance results are reported for each partitioning instance.

The solution qualities are represented by percent load imbalance values. The percent load imbalance of a partition is computed as $100 \times (B - B^*)/B^*$, where B denotes the bottleneck value of the respective partition.

Table 3.3: Percent load imbalance values for the processor speed range of 1–8 for the volume rendering dataset

CCP instance		Heuristics		OPT
Name	K	RB	MP	
blunt	32	0.27	0.31	0.08
	64	0.62	0.78	0.16
	128	1.35	2.07	0.32
	256	2.94	4.67	0.64
	512	7.27	10.96	1.27
	1024	15.15	21.94	2.83
	2048	36.90	49.23	4.99
comb	32	0.17	0.24	0.06
	64	0.44	0.63	0.11
	128	1.11	1.60	0.23
	256	2.38	3.63	0.45
	512	5.42	7.97	0.92
	1024	12.94	18.24	1.83
	2048	26.61	41.66	3.64
post	32	0.11	0.13	0.03
	64	0.25	0.39	0.07
	128	0.61	0.86	0.13
	256	1.34	2.05	0.27
	512	3.10	4.32	0.54
	1024	6.59	9.21	1.09
	2048	16.21	19.82	2.15

3.5.2 CCP Experiments

The proposed CCP algorithms were implemented in the Java language. Tables 3.3–3.6 compare the solution qualities of heuristics with respect to those of the optimal partitions obtained by the exact algorithms. In these tables, OPT values refer to the optimal load imbalance values.

Tables 3.3 and 3.4 respectively display the percent load imbalance values obtained in mapping the volume rendering and sparse matrix task chains onto processor chains with 1–8 execution speed range. As seen in these two tables, RB performs much better than MP. Out of 63 partitioning instances, RB found better solutions than MP in all but one instance.

As seen in Tables 3.3 and 3.4, in general, the quality gap between exact

Table 3.4: Percent load imbalance values for the processor speed range of 1–8 for the sparse matrix dataset

CCP instance		Heuristics		OPT
Name	K	RB	MP	
g7jac050sc	32	2.21	3.08	0.40
	64	4.88	6.06	0.75
	128	12.21	17.16	1.52
	256	29.06	42.86	3.10
	512	84.54	90.48	6.60
	1024	171.47	289.02	13.59
	2048	261.51	624.91	30.96
language	32	4.58	4.93	0.21
	64	22.60	23.06	0.40
	128	42.06	71.35	1.25
	256	98.08	184.87	35.81
	512	230.49	379.11	171.98
	1024	527.56	1,173.23	443.95
	2048	1,191.77	2,294.59	992.35
mark3jac060	32	0.32	0.54	0.08
	64	0.87	1.01	0.17
	128	2.09	2.75	0.36
	256	5.98	6.90	0.69
	512	15.47	18.17	1.36
	1024	30.23	51.57	2.89
	2048	64.50	127.93	5.92
Stanford	32	12.91	22.85	2.46
	64	42.77	84.14	5.38
	128	110.83	274.42	21.32
	256	204.46	617.98	138.66
	512	435.52	1,058.28	377.97
	1024	1,009.58	2,585.17	855.91
	2048	1,978.18	5,313.99	1,819.63
Stanford_Berkeley	32	10.76	16.91	1.40
	64	49.53	57.69	3.29
	128	89.68	177.24	8.19
	256	160.39	375.68	57.31
	512	315.61	761.14	215.05
	1024	624.98	1,911.41	530.08
	2048	1,248.18	3,949.65	1,165.31
torso1	32	1.74	2.15	0.45
	64	3.82	4.91	0.91
	128	8.75	10.30	1.84
	256	22.46	31.18	3.69
	512	31.68	75.51	7.48
	1024	75.55	75.89	17.86
	2048	252.44	252.44	27.61

Table 3.5: Percent load imbalance values for different processor speed ranges for the volume rendering dataset

CCP instance		1–4		1–8		1–16	
Name	K	RB	OPT	RB	OPT	RB	OPT
blunt	32	0.21	0.08	0.27	0.08	0.38	0.08
	64	0.39	0.16	0.62	0.16	0.93	0.16
	128	1.06	0.31	1.35	0.32	2.21	0.31
	256	2.19	0.64	2.94	0.64	5.54	0.64
	512	4.62	1.27	7.27	1.27	11.57	1.25
	1024	10.83	2.70	15.15	2.83	26.88	2.61
	2048	22.43	4.93	36.90	4.99	52.25	5.42
comb	32	0.12	0.06	0.17	0.06	0.22	0.06
	64	0.35	0.11	0.44	0.11	0.72	0.11
	128	0.77	0.23	1.11	0.23	1.65	0.23
	256	1.58	0.45	2.38	0.45	3.78	0.45
	512	3.53	0.91	5.42	0.92	9.61	0.91
	1024	7.71	1.82	12.94	1.83	19.75	1.83
	2048	17.53	3.67	26.61	3.64	44.69	3.64
post	32	0.07	0.03	0.11	0.03	0.17	0.03
	64	0.18	0.07	0.25	0.07	0.40	0.07
	128	0.40	0.14	0.61	0.13	0.91	0.13
	256	0.87	0.27	1.34	0.27	2.25	0.27
	512	1.88	0.54	3.10	0.54	4.66	0.54
	1024	4.41	1.09	6.59	1.09	11.42	1.08
	2048	8.87	2.26	16.21	2.15	26.87	2.16
Geometric averages over K							
	32	0.12	0.05	0.17	0.05	0.24	0.05
	64	0.29	0.11	0.41	0.11	0.65	0.11
	128	0.69	0.21	0.97	0.21	1.49	0.21
	256	1.44	0.43	2.11	0.43	3.61	0.43
	512	3.13	0.86	4.96	0.86	8.03	0.85
	1024	7.17	1.75	10.89	1.78	18.23	1.73
	2048	15.16	3.45	25.15	3.39	39.73	3.49

algorithms and heuristics increases with increasing number of processors. For instance, in 2048-way partitioning of the torso1 matrix, best heuristic finds a solution with 252.44% load imbalance, which means a processor is loaded more than 3.5 times the average load, causing a slowdown as the number of processors increase. An optimal solution however, will have a load imbalance value of 27.61%, providing scalability to thousands of processors.

Tables 3.5 and 3.6 display the variation of load balancing performances of

Table 3.6: Percent load imbalance values for different processor speed ranges for the sparse matrix dataset

CCP instance		1–4		1–8		1–16	
Name	K	RB	OPT	RB	OPT	RB	OPT
g7jac050sc	32	1.22	0.37	2.21	0.40	2.53	0.40
	64	3.53	0.79	4.88	0.75	6.96	0.76
	128	8.94	1.57	12.21	1.52	16.15	1.52
	256	19.62	3.18	29.06	3.10	65.36	3.16
	512	42.24	6.62	84.54	6.60	104.54	6.68
	1024	124.82	14.92	171.47	13.59	162.21	13.56
	2048	307.43	32.67	261.51	30.96	261.88	30.02
language	32	0.36	0.05	4.58	0.21	1.39	0.10
	64	14.09	0.41	22.60	0.40	6.57	0.22
	128	51.77	1.01	42.06	1.25	22.46	1.39
	256	102.08	52.24	98.08	35.81	99.07	27.82
	512	257.83	203.88	230.49	171.98	232.00	156.36
	1024	554.09	506.99	527.56	443.95	519.77	415.09
	2048	1,210.34	1,115.84	1,191.77	992.35	1,088.49	933.33
mark3jac060	32	0.27	0.08	0.32	0.08	0.40	0.08
	64	0.68	0.17	0.87	0.17	1.17	0.16
	128	1.67	0.34	2.09	0.36	3.15	0.35
	256	4.15	0.69	5.98	0.69	10.32	0.69
	512	8.82	1.38	15.47	1.36	22.87	1.40
	1024	20.17	2.85	30.23	2.89	49.73	2.82
	2048	41.26	5.82	64.50	5.92	111.65	5.68
Stanford	32	16.93	2.53	12.91	2.46	20.07	2.61
	64	42.61	5.93	42.77	5.38	48.28	4.88
	128	122.92	32.98	110.83	21.32	90.44	17.79
	256	219.75	167.53	204.46	138.66	215.16	124.62
	512	466.32	434.02	435.52	377.97	427.96	350.50
	1024	1,019.25	966.68	1,009.58	855.91	956.15	805.19
	2048	2,131.61	2,036.65	1,978.18	1,819.63	1,935.93	1,715.91
Stanford_Berkeley	32	7.14	1.29	10.76	1.40	15.32	1.44
	64	26.91	2.51	49.53	3.29	43.39	3.29
	128	85.08	8.96	89.68	8.19	74.51	8.02
	256	191.93	76.34	160.39	57.31	146.90	48.06
	512	331.15	251.99	315.61	215.05	316.54	196.95
	1024	622.85	603.10	624.98	530.08	584.74	496.65
	2048	1,339.44	1,308.36	1,248.18	1,165.31	1,261.41	1,096.94
torso1	32	1.01	0.46	1.74	0.45	1.91	0.45
	64	2.50	0.89	3.82	0.91	4.64	0.88
	128	5.82	1.72	8.75	1.84	14.14	1.85
	256	10.03	3.49	22.46	3.69	22.75	3.73
	512	16.01	5.37	31.68	7.48	65.98	8.26
	1024	40.87	13.12	75.55	17.86	186.70	15.92
	2048	96.14	38.26	252.44	27.61	231.35	32.85
Geometric averages over K							
	32	1.57	0.36	3.04	0.47	3.06	0.42
	64	6.78	0.94	9.59	0.97	8.97	0.86
	128	18.99	2.55	21.30	2.45	21.85	2.41
	256	38.99	13.12	48.21	11.44	60.30	10.51
	512	78.70	32.11	104.64	31.31	130.58	30.67
	1024	181.87	74.04	225.17	72.17	275.55	68.26
	2048	401.91	166.92	481.94	148.31	511.84	146.37

heuristics and exact algorithms with varying processor speed ranges for the volume rendering and sparse matrix task chains, respectively. Since `RB` outperforms `MP`, only the results for the `RB` heuristic are displayed in these two tables. The bottom parts of these two tables show the geometric averages of the percent load imbalance values over the number of processors.

As seen in Tables 3.5 and 3.6, in general, the performance gap between heuristics and exact algorithms decrease with decreasing processor speed range. However, there exists considerable quality difference between the heuristics and exact algorithms even for the smallest 1–4 speed range.

In constructing the processor chains for the experiments, in addition to the random processor ordering, we also investigated different orderings of the processors having the same speed. In this context, we experimented with the cases where processors having the same speed ordered consecutively, assuming that such processors belong to the same homogenous cluster and hence they are naturally adjacent to each other in the processor chain. We did not observe a considerable sensitivity of the relative load balancing performance between heuristics and exact algorithms to the ordering of processors having the same speed.

Tables 3.7–3.9 display the execution times of the proposed CCP algorithms on a workstation equipped with a 3 GHz Pentium-IV and 1 GB of memory. In these tables, `NC+`, `BID`, and `EBS` respectively represent the `NICOL+`, `BIDDING`, and `EXACT-BISECTION` presented in Algorithms 3.8, 3.9, and 3.10.

Tables 3.7 and 3.8 respectively display the execution times of the CCP algorithms for mapping the volume rendering and sparse matrix task chains onto processor chains with 1–8 execution speed range. In these two tables, relative performance comparison of heuristics shows that `MP` is slightly faster than `RB`. Since `RB` outperforms `MP` in terms of solution quality as shown in Tables 3.3 and 3.4, these results reveal the superiority of `RB` to `MP`.

In Tables 3.7 and 3.8, relative performances of exact CCP algorithms show that both `NICOL+` and `EBS` are an order of magnitude faster than `DP+` and `BID` for both volume rendering and sparse matrix datasets. As also seen in these two

Table 3.7: Partitioning times (in msecs) for the processor speed range of 1–8 for the volume rendering dataset

CCP instance		Heuristics		Exact algorithms			
Name	K	RB	MP	DP+	NC+	BID	EBS
blunt	32	0.37	0.36	1	0.58	0.52	0.49
	64	0.39	0.38	1	0.85	0.84	0.66
	128	0.44	0.42	2	1.39	1.91	1.05
	256	0.51	0.47	4	2.42	4.91	1.74
	512	0.64	0.57	14	4.68	13.97	3.28
	1024	0.89	0.76	54	8.67	43.05	6.45
	2048	1.37	1.12	201	15.27	97.54	12.09
comb	32	0.62	0.61	1	0.85	0.80	0.75
	64	0.65	0.64	1	1.15	1.17	0.96
	128	0.69	0.67	2	1.68	2.40	1.37
	256	0.77	0.74	5	2.87	6.04	2.13
	512	0.91	0.84	16	4.84	16.92	3.74
	1024	1.17	1.04	59	9.44	47.19	7.08
	2048	1.68	1.42	230	17.86	130.51	13.30
post	32	1.12	1.11	2	1.36	1.30	1.26
	64	1.15	1.14	2	1.68	1.69	1.46
	128	1.20	1.18	3	2.26	2.91	1.88
	256	1.29	1.26	6	3.52	6.54	2.82
	512	1.45	1.38	16	5.91	16.95	4.51
	1024	1.73	1.59	55	10.36	44.10	7.52
	2048	2.25	1.99	205	20.02	114.60	14.81

Table 3.8: Partitioning times (in msecs) for the processor speed range of 1–8 for the sparse matrix dataset

CCP instance		Heuristics		Exact algorithms			
Name	K	RB	MP	DP+	NC	BID	EBS
g7jac050sc	32	0.31	0.30	1	0.54	0.56	0.46
	64	0.33	0.32	1	0.83	1.08	0.65
	128	0.37	0.35	4	1.31	2.61	1.04
	256	0.44	0.40	13	2.47	7.23	1.80
	512	0.56	0.49	54	4.51	18.88	3.27
	1024	0.80	0.67	234	8.65	48.90	6.07
	2048	1.27	1.02	1730	15.06	100.99	11.96
language	32	7.80	7.80	17	8.19	9.19	8.05
	64	7.84	7.83	22	8.71	14.02	8.47
	128	7.91	7.89	56	9.88	32.63	9.33
	256	8.05	8.01	1999	11.27	8.25	10.63
	512	8.28	8.21	6298	12.38	8.55	11.73
	1024	8.70	8.57	15839	15.96	9.14	16.13
	2048	9.47	9.20	33199	21.82	10.29	20.59
mark3jac060	32	0.47	0.46	1	0.69	0.62	0.60
	64	0.49	0.48	1	0.96	0.94	0.76
	128	0.54	0.52	2	1.48	1.73	1.09
	256	0.62	0.58	7	2.43	3.55	1.78
	512	0.76	0.69	23	4.35	7.95	3.04
	1024	1.01	0.88	90	7.81	19.96	5.95
	2048	1.50	1.25	371	15.91	45.62	11.39
Stanford	32	4.98	4.97	26	5.51	25.10	5.38
	64	5.01	5.00	79	5.99	82.71	5.85
	128	5.08	5.06	841	7.09	437.39	6.67
	256	5.20	5.16	3989	8.42	3022.05	7.80
	512	5.41	5.34	9667	10.77	7524.42	10.08
	1024	5.79	5.65	22472	15.55	16580.61	14.83
	2048	6.48	6.20	49112	25.02	34629.44	23.78
Stanford_Berkeley	32	19.15	19.15	53	19.72	47.08	19.63
	64	19.20	19.18	154	20.60	140.26	20.17
	128	19.27	19.25	558	22.27	460.82	21.16
	256	19.39	19.35	4273	24.34	3722.02	22.24
	512	19.61	19.55	22065	27.82	10742.26	24.53
	1024	20.02	19.89	47607	34.03	22496.33	28.87
	2048	20.78	20.50	100548	46.18	46014.22	37.61
torso1	32	2.12	2.11	5	2.46	4.29	2.38
	64	2.14	2.13	9	2.83	8.80	2.66
	128	2.18	2.16	22	3.55	25.10	3.22
	256	2.26	2.22	83	5.03	76.26	4.45
	512	2.40	2.33	360	7.61	201.48	6.69
	1024	2.68	2.56	1566	13.00	522.08	10.65
	2048	3.24	2.98	6933	23.04	783.22	18.39

Table 3.9: Partitioning time averages (over K) of the exact CCP algorithms normalized with respect to those of the RB heuristic for different processor speed ranges

	1–4				1–8				1–16			
K	DP+	NC+	BID	EBS	DP+	NC+	BID	EBS	DP+	NC+	BID	EBS
Volume rendering dataset												
32	2	1.38	1.28	1.20	2	1.38	1.27	1.21	2	1.40	1.30	1.22
64	2	1.78	1.80	1.44	2	1.76	1.77	1.45	2	1.80	1.88	1.47
128	3	2.42	3.28	1.89	3	2.44	3.33	1.94	3	2.53	3.70	1.96
256	6	3.63	6.94	2.63	6	3.62	7.22	2.73	6	3.65	8.05	2.75
512	15	5.32	15.46	3.79	17	5.45	16.90	3.96	17	5.59	19.07	4.08
1024	43	7.66	32.01	5.21	46	7.77	37.55	5.79	47	7.78	43.59	5.87
2048	114	10.18	53.81	6.95	123	10.15	65.70	7.68	129	10.73	86.03	7.75
Sparse matrix dataset												
32	3	1.25	2.33	1.15	3	1.26	2.30	1.18	3	1.28	2.67	1.17
64	6	1.50	5.34	1.31	6	1.52	5.77	1.34	6	1.54	5.90	1.36
128	34	1.93	24.89	1.59	37	1.93	22.48	1.66	35	2.01	23.37	1.69
256	212	2.58	122.47	2.01	217	2.69	136.83	2.17	219	2.68	147.12	2.12
512	650	3.51	277.96	2.64	649	3.65	340.06	2.89	638	3.75	389.97	2.90
1024	1,422	4.69	565.27	3.51	1,464	4.94	701.30	3.92	1,471	5.07	812.36	3.89
2048	3,136	6.02	1,051.74	4.47	3,243	6.36	1,301.49	5.04	3,234	6.70	1,550.64	5.10

tables, EBS is slightly faster than NICOL+.

It is worth highlighting that for small to medium concurrency, the time EBS and NICOL+ algorithms take to find optimal solutions is less than three times the time of the fastest heuristic. More precisely, on overall average, EBS takes only 147% more time than the fastest heuristic for 256-way partitioning. On the other hand, at higher number of processors, the solution qualities of heuristics degrade significantly: on overall average, optimal solutions provide 5.35, 5.47 and 6.00 times better load imbalance values than the best heuristic for 512, 1024 and 2048-way partitionings, respectively. According to these findings, we recommend the use of exact CCP algorithms instead of heuristics for heterogeneous systems.

Table 3.9 displays the variation of running time performances of the CCP algorithms with varying processor speed ranges for the volume rendering and sparse matrix task chains. For a better performance comparison, execution times of the algorithms were normalized with respect to those of the RB heuristic and averages of these normalized values over K are presented in the table. We should mention here that the running time of the RB heuristic does not change with varying processor speed range, as expected. As seen in Table 3.9, notable performance variation occurs only for the BIDDING algorithm whose running time generally increases with increasing processor speed range.

Table 3.10: Geometric averages (over K) of percent load imbalance values for R randomly ordered processor chains for the volume rendering dataset with the processor speed range of 1–8

	$R = 10$		$R = 100$		$R = 1000$		$R = 10000$	
K	best	avg	best	avg	best	avg	best	avg
32	0.042	0.050	0.038	0.049	0.036	0.049	0.033	0.048
64	0.097	0.111	0.091	0.112	0.082	0.112	0.077	0.112
128	0.199	0.217	0.189	0.219	0.176	0.218	0.172	0.219
256	0.402	0.427	0.391	0.430	0.377	0.428	0.370	0.428
512	0.852	0.870	0.823	0.870	0.807	0.868	0.791	0.868
1024	1.787	1.849	1.750	1.856	1.727	1.855	1.719	1.855
2048	3.337	3.414	3.245	3.401	3.159	3.402	3.150	3.401

3.5.3 CP Experiments

Tables 3.10 and 3.11 display the results of our experiments to show the sensitivity of the solution quality of CP problem instances to the processor orderings for the processor speed range of 1–8. In these experiments, we find the optimal CCP solutions for R randomly ordered processor chains of a CP instance, and display geometric averages of the best and average load imbalance values over number of processors. As seen in the tables, for a fixed K, the average imbalance values almost remain the same for different values of R. Although the best imbalance values decrease with increasing R, the decreases are quite small, especially for large K. Moreover, for a fixed R, the relative difference between the best and average imbalance values decreases with increasing K.

These experimental findings show that processor ordering has only a minor effect on solution quality. This is expected since the variance among processor speeds is low, unlike the variance among task weights. Therefore, using an exact CCP algorithm on a number of randomly permuted processor chains can serve as an effective heuristic for the CP problem.

Table 3.12 displays the results of our experiments to show the sensitivity of the solution quality of CP problem instances to the processor speed range. In these experiments, for each CP instance, we find the optimal CCP solutions

Table 3.11: Geometric averages (over K) of percent load imbalance values for R randomly ordered processor chains for the sparse matrix dataset with the processor speed range of 1–8

	$R = 10$		$R = 100$		$R = 1000$		$R = 10000$	
K	best	avg	best	avg	best	avg	best	avg
32	0.133	0.483	0.104	0.656	0.068	0.588	0.057	0.534
64	0.460	0.906	0.313	0.835	0.257	0.924	0.222	0.935
128	1.304	2.526	1.216	2.484	1.124	2.462	1.020	2.573
256	10.843	11.411	10.291	11.420	10.127	11.427	9.958	11.433
512	31.153	31.694	29.385	31.776	29.078	31.747	28.922	31.735
1024	70.403	71.296	69.160	71.540	68.472	71.530	67.855	71.521
2048	147.792	150.082	146.616	150.360	143.709	150.191	142.917	150.283

for $R = 10000$ randomly ordered processor chains, and display the best load imbalance value. As seen in Table 3.12, we do not observe a considerable sensitivity of the solution quality of the CP problem instances to the procesor speed range. Notable sensitivity is observed only for the `language`, `Stanford`, and `Stanford_Berkeley` sparse matrix datasets, which have high task weight variation (i.e., large $w_{\max}/w_{\text{avg}}$ value). In these datasets, load imbalance values decrease with increasing processor speed range, which possibly because the adverse effect of tasks with large weight on load imbalance can be more easily resolved by mapping them to the processors with larger execution speed.

Table 3.12: Best percent load imbalance values for $R = 10000$ randomly ordered processor chains with different processor speed ranges

Volume rendering dataset					Sparse matrix dataset				
CCP instance					CCP instance				
Name	K	1–4	1–8	1–16	Name	K	1–4	1–8	1–16
blunt	32	0.029	0.053	0.051	g7jac050sc	32	0.154	0.146	0.092
	64	0.125	0.134	0.117		64	0.390	0.366	0.371
	128	0.207	0.267	0.241		128	1.003	1.016	0.994
	256	0.628	0.559	0.528		256	2.402	2.226	2.439
	512	1.055	1.193	1.157		512	5.493	5.497	5.297
	1024	2.300	2.992	2.543		1024	13.187	11.727	11.829
	2048	5.000	4.554	4.938		2048	28.115	28.269	26.974
comb	32	0.037	0.034	0.034	language	32	0.004	0.003	0.004
	64	0.076	0.075	0.079		64	0.011	0.010	0.013
	128	0.183	0.180	0.179		128	0.052	0.050	0.040
	256	0.377	0.387	0.380		256	55.560	34.304	24.151
	512	0.818	0.814	0.812		512	206.845	168.371	151.509
	1024	1.707	1.662	1.694		1024	511.078	443.036	407.589
	2048	3.561	3.508	3.522		2048	1,122.157	977.521	915.654
post	32	0.020	0.020	0.020	mark3jac060	32	0.033	0.039	0.041
	64	0.048	0.046	0.047		64	0.095	0.104	0.103
	128	0.109	0.107	0.108		128	0.245	0.232	0.245
	256	0.233	0.234	0.230		256	0.536	0.547	0.544
	512	0.466	0.510	0.479		512	1.173	1.154	1.215
	1024	0.948	1.022	0.988		1024	2.501	2.474	2.504
	2048	2.240	1.957	2.043		2048	5.516	5.255	5.225
					Stanford	32	0.239	0.127	0.128
						64	0.960	0.889	0.525
						128	35.643	12.897	14.879
						256	173.373	136.019	118.176
						512	439.233	371.620	341.987
						1024	973.874	854.300	792.008
						2048	2,047.748	1,793.575	1,684.852
					Stanford_Berkeley	32	0.047	0.063	0.073
						64	0.740	0.554	0.666
						128	2.831	3.307	2.843
						256	80.192	55.570	43.809
						512	255.431	210.865	191.333
						1024	607.837	529.020	487.961
						2048	1,315.674	1,148.137	1,076.473
					torso1	32	0.315	0.229	0.307
						64	0.771	0.639	0.677
						128	1.112	2.240	1.538
						256	1.890	3.087	3.004
						512	4.859	6.996	8.198
						1024	12.046	16.806	15.439
						2048	38.975	28.495	31.079

Chapter 4

Independent Task Assignment: Improving Performances of Well-known Constructive Heuristics

In this chapter, we present our studies on improving existing independent task assignment heuristics, `MinMin`, `MaxMin`, `Suff`, and `GA`. The sections of this chapter are organized as follows. Table 4.1 summarizes the notation used in this chapter. Section 4.1 describes the existing algorithms. The proposed `MinMin+` is presented in Section 4.2. Improved `MaxMin+` algorithm is described in Section 4.3. `Suff+` algorithm is in Section 4.4, and the improved `GA` algorithm is discussed in Section 4.5. We present our experiment results in Section 4.6.

4.1 Existing Algorithms

MinMin: The `MinMin` heuristic [63] proceeds in N iterations for the assignment of N independent tasks to K processors. At each iteration, a previously unassigned task is selected and assigned to a processor. The selected task is removed

Table 4.1: The notation used in this chapter

Notation	Explanation
A	task-to-processor assignment vector
E	expected-time-compute matrix
G	number of chromosomes used in the GA algorithm
H	number of iterations of the GA algorithm
K	number of processors
M	makespan
M^*	ideal makespan
N	number of tasks
$\mathcal{P}$	set of processors
P_k	kth processor
Q_k	priority queue of P_k in the MinMin+ algorithm
R	machine heterogeneity constant
$\mathcal{T}$	set of tasks
T_i	ith task
U	a set of tasks
e_k	current load of processor P_k
i, j	indices that refer to tasks
k, ℓ	indices that refer to processors
m	number of MaxMin-based assignments in MaxMin+
$x_{i,k}$	computation cost of task T_i on processor P_k
α	exponent constant for power-law distribution
γ	relative cost for the RC algorithm

Algorithm 4.1 MinMinSelect(U, e, x, K)

```
1: min' ← ∞
2: for each i ∈ U do
3:     min ← ∞
4:     for k ← 1 to K do
5:         if e_k + x_{i,k} < min then
6:             min ← e_k + x_{i,k}
7:             kmin ← k
8:     if min < min' then
9:         min' ← e_kmin + x_{i,kmin}
10:        k' ← kmin
11:        i' ← i
12: return ⟨i', k'⟩
```

from further consideration in the remaining iterations. The task-to-processor assignment in each iteration is decided based on a two-step procedure. In the first step, `MinMin` computes the minimum completion time (MCT) of each unassigned task over the processors to find the best processor, which can complete the processing of that task at earliest time. This decision is made taking into account the current loads of processors (e_k) and the execution time of the task on each processor ($x_{i,k}$). In the second step, `MinMin` selects the task with the minimum MCT among all unassigned tasks and assigns the task to its best processor found in the first step. Due to the task selection policy adopted in the second step, `MinMin` favors the assignment of tasks with lower costs in earlier iterations, and hence the assignment of tasks with higher costs are usually performed during the later iterations. The two-step selection algorithm is provided in Algorithm 4.1. An $O(KN^2)$-time algorithm for `MinMin` is given in Algorithm 4.2.

MaxMin: `MaxMin` [5,10,48,63] differs from `MinMin` in the task selection policy adopted in the second step of the task-to-processor assignment procedure. Unlike `MinMin`, which selects the task with the minimum MCT, `MaxMin` selects the task with the maximum MCT and then assigns it to the best processor found in the first step (Algorithm 4.3). Due to this task selection policy, `MaxMin` performs the assignment of tasks with higher costs in earlier iterations. The algorithm for `MaxMin` is presented in Algorithm 4.4.

Algorithm 4.2 MinMin(x, K, N)

```
1: U ← {1, 2, ..., N}
2: for k ← 1 to K do
3:     e_k ← 0
4: while U is not empty do
5:     ⟨i', k'⟩ ← MinMinSelect(U, e, x, K)
6:     A[i'] ← k'
7:     e_k' ← e_k' + x_{i',k'}
8:     U ← U − {i'}
9: return A
```

Algorithm 4.3 MaxMinSelect(U, e, x, K)

```
1: max ← 0
2: for each i ∈ U do
3:     min ← ∞
4:     for k ← 1 to K do
5:         if e_k + x_{i,k} < min then
6:             min ← e_k + x_{i,k}
7:             kmin ← k
8:     if min > max then
9:         max ← e_kmin + x_{i,kmin}
10:        k' ← kmin
11:        i' ← i
12: return ⟨i', k'⟩
```

Algorithm 4.4 MaxMin(x, K, N)

```
1: U ← {1, 2, ..., N}
2: for k ← 1 to K do
3:     e_k ← 0
4: while U is not empty do
5:     ⟨i', k'⟩ ← MaxMinSelect(U, e, x, K)
6:     A[i'] ← k'
7:     e_k' ← e_k' + x_{i',k'}
8:     U ← U − {i'}
9: return A
```

Algorithm 4.5 RASA(x, K, N)

```
1:  U ← {1, 2, ..., N}
2:  for k ← 1 to K do
3:      e_k ← 0
4:  for r ← 1 to N do
5:      if r is odd then
6:          ⟨i', k'⟩ ← MaxMinSelect(U, e, x, K)
7:      else
8:          ⟨i', k'⟩ ← MinMinSelect(U, e, x, K)
9:      A[i'] ← k'
10:     e_k' ← e_k' + x_{i',k'}
11:     U ← U − {i'}
12: return A
```

RASA: In [90], the drawbacks of `MaxMin` and `MinMin` are analyzed and a hybrid algorithm, referred to as `RASA`, is proposed. `RASA` alternates between `MaxMin` and `MinMin` in its iterations. In particular, `MaxMin` is used in odd rounds while `MinMin` is used in even rounds. The `RASA` algorithm, which runs in $O(KN^2)$ time, is displayed in Algorithm 4.5.

Sufferage: The main difference between `Suff` [78] and `MinMin` is the task selection policy. In the first step of the process, `Suff` computes the second MCT value in addition to the MCT value for each task. In the second step, the sufferage value, which is defined as the difference between the MCT and the second MCT values of a task, is taken into account. `Suff` selects the task with the largest sufferage and assigns it to the best processor found in the first step. The algorithm for `Suff` is presented in Algorithm 4.7.

Relative Cost (RC): `RC` [107] is a constructive heuristic similar to `MinMin`, but it uses a different selection criterion which does not lead to a bias between small tasks and large tasks. At each iteration of the algorithm, `RC` selects the task with the lowest relative cost, which is calculated as

$$\gamma = \left(\frac{\min_k \{x_{i,k} + e_k\}}{\operatorname{avg}_k \{x_{i,k} + e_k\}} \right) + \left(\frac{x_{i,k^*(i)}}{\operatorname{avg}_k \{x_{i,k}\}} \right)^{\xi}, \tag{4.1}$$

Algorithm 4.6 SUFFSELECT(U, x, K, N)

```
1: sufferage' ← 0
2: for each i ∈ U do
3:     min ← ∞
4:     second_min ← ∞
5:     for k ← 1 to K do
6:         if e_k + x_{i,k} < min then
7:             second_min ← min
8:             min ← e_k + x_{i,k}
9:             kmin ← k
10:        else if e_k + x_{i,k} < second_min then
11:            second_min ← e_k + x_{i,k}
12:    sufferage ← second_min − min
13:    if sufferage > sufferage' then
14:        sufferage' ← sufferage
15:        k' ← kmin
16:        i' ← i
17: return ⟨i', k'⟩
```

Algorithm 4.7 SUFF(x, K, N)

```
1: U ← {1, 2, . . . , N}
2: for k ← 1 to K do
3:     e_k ← 0
4: while U is not empty do
5:     ⟨i', k'⟩ ← SUFFSELECT(U, e, x, K)
6:     A[i'] ← k'
7:     e_{k'} ← e_{k'} + x_{i',k'}
8:     U ← U − {i'}
9: return A
```

Algorithm 4.8 RC(x, K, N)

1: $U \leftarrow \{1, 2, \ldots, N\}$
2: **for** $k \leftarrow 1$ **to** K **do**
3: $\quad e_k \leftarrow 0$
4: **for** $i \leftarrow 1$ **to** N **do**
5: $\quad avg \leftarrow \text{avg}_k\{x_{i,k}\}$
6: $\quad$ **for** $k \leftarrow 1$ **to** K **do**
7: $\quad\quad \gamma_s[i,k] \leftarrow x_{i,k}/avg$
8: **for** $j \leftarrow 1$ **to** N **do**
9: $\quad \gamma_min \leftarrow \infty$
10: $\quad$ **for** $i \leftarrow 1$ **to** N **do**
11: $\quad\quad avg \leftarrow \text{avg}_k\{e_k + x_{i,k}\}$
12: $\quad\quad k \leftarrow \text{argmin}_k\{e_k + x_{i,k}\}$
13: $\quad\quad min \leftarrow x_{i,k}$
14: $\quad\quad \gamma \leftarrow min/avg \times \gamma_s[i,k]^{\xi}$
15: $\quad\quad$ **if** $\gamma < \gamma_min$ **then**
16: $\quad\quad\quad \gamma_min \leftarrow \gamma$
17: $\quad\quad\quad i' \leftarrow i$
18: $\quad\quad\quad k' \leftarrow k$
19: $\quad A[i'] \leftarrow k'$
20: $\quad e_{k'} \leftarrow e_{k'} + x_{i',k'}$
21: $\quad U \leftarrow U - \{i'\}$
22: **return** A

where $k^*(i) = \text{argmin}_k \{x_{i,k} + e_k\}$ in the current iteration. The selected task is assigned to processor $k^*(i)$. ξ is a parameter in the $[0, 1]$ range and is used to control the effects of the first and second terms in Eq. 4.1. `RC` is reported as a high-quality algorithm and runs in $O(KN^2)$ time. The `RC` algorithm is displayed in Algorithm 4.8.

Genetic Algorithm (GA): `GA` [10,105] is an example of more complex algorithms that use `MinMin` as a component. `GA` uses `MinMin` as an initial chromosome and improves the solution of `MinMin` using genetic algorithm techniques. In this approach, each chromosome represents a different task-to-processor assignment. Assuming G chromosomes, one of the chromosomes is initially populated with `MinMin` while the remaining $G - 1$ chromosomes are populated with random assignments. Maintaining the best assignment (elitism) guarantees that the solution quality of `GA` is not worse than the quality of `MinMin`. Crossover is implemented as a single random cross on the paired chromosomes. Mutation is defined as reassigning a random task to a random processor. The initial population runs in $O(KN^2 + G \log G + NG)$ time. Each iteration of `GA` runs in $O(NG + G^2)$ time. Hence, `GA` runs in $O(KN^2 + HNG + HG^2)$ time, where H is the number of iterations.

4.2 MinMin+

The high running time complexity of the `MinMin` algorithm stems from the $O(KN)$-time cost that is incurred while computing the MCT values for every unassigned task and processor pair. Note that the MCT values and the best processor of an unassigned task may change at each iteration of the loop in Algorithm 4.2. This is because the $e_k + x_{i,k}$ value associated with an unassigned task T_i and processor P_k may change as the e_k values are updated throughout the iterations. Without any loss of generality, let us assume that a task is assigned to a processor P_k in the previous iteration. This assignment increases the e_k value. Therefore, in the next iteration, the $e_k + x_{i,k}$ values for all unassigned tasks need to be recomputed for processor P_k. This task-oriented view of the

`MinMin` algorithm forms a lower bound of $\Omega(KN^2)$ on the running time of the algorithm.

In this work, we demonstrate that the above-mentioned quadratic lower bound can be avoided by switching from the task-oriented view to a processor-oriented view. To this end, we propose a novel algorithm, referred to as `MinMin+`. In this algorithm, the MCT values that are associated with each processor are separately maintained, instead of being unnecessarily recomputed at each iteration for every unassigned task. In particular, we use a priority queue Q_k for each processor P_k to maintain the completion times of all tasks on that processor. More specifically, each task T_i is maintained in K different priority queues, keyed by their $x_{i,k}$ values. Each priority queue Q_k supports the MIN, DELETE, and BUILD operations. MIN(Q_k) is a query operation that returns the id of the unassigned task that has the minimum completion time on processor P_k. DELETE(Q_k, i) is an update operation that removes task T_i from Q_k. The BUILD(k) operation initializes the data structures. We also maintain a boolean array F of size N. Each array element $F[i]$ indicates whether task T_i is yet assigned to a processor or not. Initially, we set all $F[i]$ values to FALSE since no task is assigned to a processor at the beginning.

The proposed `MinMin+` algorithm is given in Algorithm 4.9. The `MinMin+Init` function (Algorithm 4.10) is called in the first line of the algorithm to perform the necessary initializations. The following main loop (lines 2–8) performs N iterations, assigning a task to a processor at each iteration. The `MinMin+Select` function (Algorithm 4.11) invokes a MIN(Q_k) operation on each priority queue Q_k to find a candidate task for processor P_k. The candidate task T_i selected for processor P_k is effectively the task that will increase the current completion time of P_k (i.e., e_k) by the smallest amount if T_i is assigned to P_k. For each processor P_k, the execution time of the candidate task T_i on P_k is added to e_k to compute the updated e_k value for P_k if T_i is assigned to P_k. A running-min operation performed over these K updated e_k values gives the minimum MCT value (min) for the current iteration as well as the task-to-processor assignment (i', k') that achieves this minimum MCT value. At the end of each iteration of the main loop, the assigned task $T_{i'}$ is deleted from all priority queues (lines 7 and 8).

Algorithm 4.9 MINMIN+(x, K, N)

```
1: ⟨e, F, Q⟩ ← MinMin+Init(x, K)
2: for j ← 1 to N do
3:     ⟨i', k'⟩ ← MinMin+Select(Q, e, K)
4:     A[i'] ← k'
5:     e_{k'} ← e_{k'} + x_{i',k'}
6:     F[i'] ← TRUE
7:     for k ← 1 to K do
8:         Delete(Q_k, i')
9: return A
```

Algorithm 4.10 MINMIN+INIT(x, K)

```
1: for k ← 1 to K do
2:     e_k ← 0
3: for i ← 1 to N do
4:     F[i] ← FALSE
5: for k ← 1 to K do
6:     Q_k ← Build(k)                    ▷ Q_k contains records of ⟨i, x_{i,k}⟩ .
7: return ⟨e, F, Q⟩
```

For the implementation of the priority queue, we have considered two alternatives: binary heap and sorted linear array. Although both implementations lead to the same worst-case running time complexity, our empirical results indicate that the sorted linear array implementation yields significantly lower execution times compared to the binary-heap implementation. Hence, in what follows, we present the running time analysis of the `MinMin+` algorithm only for the sorted linear array implementation.

In the sorted linear array implementation, for each processor P_k, we maintain a linear array Q_k, which contains N tuples of the form $\langle i, x_{i,k} \rangle$. The BUILD operation sorts the tuples in Q_k in increasing order of the $x_{i,k}$ values. For each Q_k, we maintain an index b_k, indicating the unassigned task that currently has the smallest completion time on processor P_k. The BUILD operation initializes the b_k value to 1. The overall running time of the BUILD operation is $O(N \log N)$. The MIN(Q_k) operation can be realized in $O(1)$ time, simply by returning the

Algorithm 4.11 MinMin+Select(Q, e, K)

1: $min \leftarrow \infty$
2: **for** $k \leftarrow 1$ **to** K **do**
3: $\quad \langle i, x \rangle \leftarrow$ Min(Q_k)
4: $\quad$ **if** $e_k + x < min$ **then**
5: $\quad\quad min \leftarrow e_k + x$
6: $\quad\quad k' \leftarrow k$
7: $\quad\quad i' \leftarrow i$
8: **return** $\langle i', k' \rangle$

task id of the b_k-th tuple in Q_k. After a task T_i is assigned to a processor, it is deleted by setting $F[i]$ to TRUE and running a DELETE(Q_k) operation on every Q_k. Since $Q_k[1, \ldots, b_k - 1]$ contains the tasks that are already assigned, the DELETE(Q_k) operation can be realized by advancing the b_k index on Q_k until an unassigned task is encountered. Although the worst-case running time of an individual DELETE(Q_k) operation is $O(N)$, the amortized cost of DELETE(Q_k) operation is $O(1)$. This is because N DELETE operations performed on Q_k can lead to at most N increments on b_k. This simple yet efficient implementation of the DELETE operation makes the sorted linear array implementation preferable over the binary heap implementation. The proposed `MinMin+` algorithm involves K BUILD(k), $K \times N$ MIN(Q_k), and $K \times N$ DELETE(Q_k) operations. Hence, the overall running time complexity is $O(KN \log N + KN + KN) = O(KN \log N)$.

4.3 MaxMin+

In some problem instances, the task sizes follow a power-law distribution, i.e., there are a small number of very large tasks and a very large number of small tasks. In such cases, the assignment of large tasks can have a significant impact on the load of the most heavily loaded processor (i.e., makespan) and determine the resulting solution quality. In case of the `MinMin` heuristic, due to the adopted task selection policy, smaller tasks are assigned in earlier iterations, delaying the assignment of larger tasks to later iterations. The solution quality obtained in

the earlier iterations is likely to deteriorate due to the late assignment of very large tasks. In case of the `MaxMin` heuristic, the larger tasks are assigned in earlier iterations, but not necessarily to their favorite processors. To demonstrate the issue, let us consider the first few iterations of `MaxMin`. The first iteration assigns the largest task to its favorite processor. Let us assume that the second largest task has the same favorite processor as the largest task. In the second iteration, the task selection policy of `MaxMin` prevents the assignment of the second largest task to its favorite processor. In the next iteration, the third largest task loses the flexibility of being assigned to the favorite processors of the largest two tasks and so on.

To alleviate the above-mentioned drawbacks of the `MinMin` and `MaxMin` heuristics, we combine these two heuristics under a hybrid heuristic, which we refer to as `MaxMin+`. Like `MinMin` and `MaxMin`, the `MaxMin+` heuristic involves a main loop that assigns a selected task to a processor at each iteration. Within an iteration, the heuristic first computes a task-to-processor assignment according to the `MinMin` heuristic. The computed assignment is realized only if it does not lead to an increase in the makespan of the previous iteration. If, however, the computed assignment increases the makespan, the task-to-processor assignment is recomputed according to the `MaxMin` heuristic.

The `MaxMin+` algorithm is presented in Algorithm 4.12, using the asymptotically faster `MinMin+` algorithm proposed in Section 4.2 instead of the standard `MinMin` algorithm. In the algorithm, `MinMin+Init` (line 3) performs the necessary initializations as in `MinMin+`. Line 5 computes the task-to-processor assignment according to `MinMin+`. The if statement at line 6 checks whether the computed assignment increases the current makespan. Line 7 computes the task-to-processor assignment according to `MaxMin`.

As described in Section 4.1, the `RASA` heuristic also combines `MinMin` and `MaxMin`. In `RASA`, `MinMin` is executed in odd-numbered iterations while `MaxMin` is executed at even-numbered iterations. The proposed `MaxMin+` heuristic differs from `RASA` in that the choice between `MinMin` and `MaxMin` at each iteration is made in an adaptive manner, considering the current processor loads. The experimental

Algorithm 4.12 MaxMin+(x, K, N)

```
1: U ← {1, 2, ..., N}
2: makespan ← 0
3: ⟨e, F, Q⟩ ← MinMin+Init(x, K)
4: while U is not empty do
5:     ⟨i', k'⟩ ← MinMin+Select(Q, e, K)
6:     if e_{k'} + x_{i',k'} > makespan then
7:         ⟨i', k'⟩ ← MaxMinSelect(U, e, x, K)
8:         makespan ← e_{k'} + x_{i',k'}
9:     A[i'] ← k'
10:    e_{k'} ← e_{k'} + x_{i',k'}
11:    U ← U − {i'}
12:    F[i'] ← TRUE
13:    for k ← 1 to K do
14:        Delete(Q_k, i')
15: return A
```

results reported in Section 4.6 shows the success of this adaptive policy with respect to the policy adopted in `RASA`.

The running time of `MaxMin+` depends on the frequency of `MaxMin`-based assignments. In practice, `MaxMin+` is expected to run slower than `MinMin+` since line 7 is executed when the assignment is performed according to `MaxMin`. `MaxMin+` is expected to run faster than `MaxMin`. The performance of `MaxMin+` depends on the ratio of the `MaxMin`-based assignments to the total number of assignments.

In the following lemmas, we describe the theoretical behavior of the `MaxMin+` algorithm and find the expected number of `MaxMin`-based assignments for some statistical distributions.

Lemma 4.3.1 `MaxMin+` *makes one* `MaxMin`*-based assignment in the best case, and makes N* `MaxMin`*-based assignments in the worst case.*

Proof: Consider the first iteration, $\forall k, e_k = 0$ and hence *makespan* $= \max_k e_k = 0$. Let $\langle i', k' \rangle$ be the selection according to `MinMin+Select`. Since

$x_{i',k'} > 0$, $e_{k'} + x_{i',k'} > 0 = \mathit{makespan}$. Thus, the selection will increase makespan and, at least in the first iteration, `MaxMin`-based assignment will be used. Thus, the number of `MaxMin`-based assignments is at least 1. To show that in the best case the number of `MaxMin`-based assignments is 1, we will construct an example that requires no other `MaxMin`-based assignments. Let $x_{i,k}$ be provided for $1 \leq i < N, 1 \leq k \leq K$. We will set the remaining values of the ETC matrix $(x_{N,1}, \ldots, x_{N,K})$ as

$$x_{N,1} = \sum_{1 \leq i < N} \max_{1 \leq k \leq K} x_{i,k}, \tag{4.2}$$

$$x_{N,k} = k + x_{N,1}, \text{for } 1 < k \leq K. \tag{4.3}$$

We are sure that, in the first iteration, `MaxMin`-based assignment will be used. For task T_N,

$$\min_{1 \leq k \leq K} x_{N,k} = x_{N,1} \tag{4.4}$$

$$= \sum_{1 \leq i < N} \max_{1 \leq k \leq K} x_{i,k} \tag{4.5}$$

$$> \max_{1 \leq i < N} \max_{1 \leq k \leq K} x_{i,k} \tag{4.6}$$

$$\geq \max_{1 \leq i < N} \min_{1 \leq k \leq K} x_{i,k}. \tag{4.7}$$

Hence, $\langle N, 1 \rangle$ will be selected in the first iteration.

In the remaining iterations, the load e_k on a processor P_k, $k > 1$ will be bounded by

$$e_k \leq \sum_{1 \leq i < N} \max_{1 < k \leq K} x_{i,k} \tag{4.8}$$

$$= x_{N,1}. \tag{4.9}$$

The e_k values for the remaining processors will never be greater than $x_{N,1}$. The makespan will never change after the first iteration and only one `MaxMin`-based assignment will be performed for this problem instance.

We will construct another problem instance to prove that the number of `MaxMin`-based assignments can be as high as N. Let $x_{i,1}$ be provided for $1 \leq i \leq N$. We will set the remaining $x_{i,k}$ values, for $1 < k \leq K$, as

$$x_{i,k} = \sum_{1 \leq j \leq N} x_{j,1}. \tag{4.10}$$

For this problem instance, always the first processor will be selected by both `MinMinSelect` and `MaxMinSelect` algorithms. Thus, at each iteration, the makespan will increase and `MaxMin`-based assignment will be used. For this problem instance, N `MaxMin`-based assignments will be performed. ■

Lemma 4.3.2 `MaxMin+` *runs in* $O(KN \log N + KNm)$ *time, where* m *is the number of* `MaxMin`*-based assignments.*

Proof: Excluding the if-block at line 6, the `MaxMin+` algorithm is almost the same as `MinMin+` and runs in $O(KN \log N)$ time. The `MaxMinSelect` algorithm runs in $O(KN)$ time. If the number of `MaxMin`-based assignments is m, then the if-block will be executed m times, and the overall cost of the block will be $O(KNm)$ time. Thus, `MaxMin+` will run in $O(KN \log N + KNm)$ time. ■

In general, the number of `MaxMin`-based assignments is expected to decrease with both increasing heterogeneity and increasing K. The former expectation is due to the higher variation in task execution costs with increasing heterogeneity, which generally results in an increase in the ratio between the weights of larger tasks and smaller tasks. Hence, a `MaxMin`-based assignment of a large task will be amortized by a large number of `MinMin`-based assignments of smaller tasks. The latter expectation is due to the extra processing power provided by the additional processors, which results in more room for the `MinMin` selections until the makespan changes. The experimental results reported in Section 4.6.2.1 support this expectation.

We present the following theorems for the special and possibly the worst case of $K = 2$ homogenous processors.

Theorem 4.3.1 *For* $K = 2$ *homogenous processors, if the task weights of a dataset have a power-law distribution with the probability density function* $f(x) = Cx^{-\alpha}$ *for* $x > x_{\min}$ *and* $\alpha > 2$*, the expected number of* `MaxMin`*-based assignments is* $\left(\frac{1}{2}\right)^{\frac{\alpha-1}{\alpha-2}} N$*.*

Proof: Without loss of generality, assume that $\langle x_1, x_2, \ldots, x_N \rangle$ is already

sorted in increasing order, i.e., $x_1 \leq x_2 \leq \ldots \leq x_N$. In the first iteration, `MaxMin`-based assignment is used. `MinMin`-based assignments are utilized until the selections are leveled against the first `MaxMin` selection, i.e., j `MinMin`-based assignments are used, where j is the maximum number that satisfies $x_N \geq \sum_{i \leq j} x_i$. After leveling, another `MaxMin` selection is used and `MinMin`-based assignments are used until the bottleneck is leveled.

Assume that M is the number of `MinMin`-based assignments. The minimum number of `MaxMin`-based assignments is used when all `MaxMin`-based assignments select the same processor. In that case, $\sum_{i>M} x_i \geq \sum_{i \leq M} x_i$. Thus, we are to find the index M such that the sum of the smallest M elements is equal to half of the sum of total values.

The expected index M can be determined by the help of Lorenz curves [76]. Lorenz curve $L(F) : [0, 1] \to [0, 1]$ is a function that takes a parameter $F \in [0, 1]$, the ratio of total distribution, and returns the expected ratio of the sum of the smallest F of total values. For example, if $L(0.8) = 0.2$, then the smallest 80% elements are expected to have a sum equal to the 20% of total values. For a given distribution, the Lorenz curve $L(F)$ can be written in terms of a probability density function $f(x)$ of the distribution as

$$L(F) = \frac{\int_{x_{\min}}^{x(F)} x f(x)\,\mathrm{d}x}{\int_{x \geq x_{\min}} x f(x)\,\mathrm{d}x} = \frac{\int_0^F x(t)\,\mathrm{d}t}{\int_0^1 x(t)\,\mathrm{d}t}, \tag{4.11}$$

where $x(F)$ is the inverse of the cumulative density function. The power-law distribution defines

$$x(F) = \frac{x_{\min}}{(1-F)^{1/(\alpha-1)}}. \tag{4.12}$$

The Lorenz curve is then calculated as

$$L(F) = \frac{\int_0^F \frac{x_{\min}}{(1-t)^{1/(\alpha-1)}}\,\mathrm{d}t}{\int_0^1 \frac{x_{\min}}{(1-t)^{1/(\alpha-1)}}\,\mathrm{d}t} \tag{4.13}$$

$$L(F) = \frac{-x_{\min}(1-t)^{(\alpha-2)/(\alpha-1)}\big|_{t=0}^{F}}{-x_{\min}(1-t)^{(\alpha-2)/(\alpha-1)}\big|_{t=0}^{1}} \tag{4.14}$$

For $1 < \alpha < 2$, this function degenerates to

$$L(F) = \begin{cases} 0 & \text{for } 0 \leq F < 1 \\ 1 & \text{for } F = 1 \end{cases} \tag{4.15}$$

Thus, for $1 < \alpha < 2$, the largest value is expected to be much larger than the sum of all other values, and only one MaxMin-based assignment is sufficient during the execution.

For $\alpha > 2$,

$$L(F) = \frac{x_{\min} - x_{\min}(1-F)^{(\alpha-2)/(\alpha-1)}}{x_{\min}} \tag{4.16}$$

$$L(F) = 1 - (1-F)^{(\alpha-2)/(\alpha-1)} \tag{4.17}$$

Thus, we are to find the F value such that L(F) = 0.5, and F can be evaluated as

$$F = 1 - \left(\frac{1}{2}\right)^{\frac{\alpha-1}{\alpha-2}}. \tag{4.18}$$

Thus, the expected number of MaxMin-based assignments is

$$(1-F) \times N = \left(\frac{1}{2}\right)^{\frac{\alpha-1}{\alpha-2}} N. \tag{4.19}$$

■

Note that, if α gets closer to 2, the number of MaxMin-based assignments decreases.

Theorem 4.3.2 *For $K = 2$ homogenous processors, if the task weights of a dataset are uniformly distributed between $x_{\min}$ and $x_{\max}$, the expected number of* MaxMin*-based assignments is $\frac{2r-\sqrt{2r^2+2}}{2r-2}N$, where $r = x_{\max}/x_{\min}$.*

Proof: The proof is similar to the proof of Theorem 4.3.1. The cumulative density function for uniform distribution is given by

$$F = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{4.20}$$

and the inverse of cumulative density function for uniform distribution can be derived as

$$x(F) = F(x_{\max} - x_{\min}) + x_{\min}. \tag{4.21}$$

The Lorenz curve is then calculated as

$$L(F) = \frac{\int_0^F x(t)\,\mathrm{d}t}{\int_0^1 x(t)\,\mathrm{d}t} \tag{4.22}$$

$$L(F) = \frac{\int_0^F t(x_{\max} - x_{\min}) + x_{\min} \,\mathrm{d}t}{\int_0^1 t(x_{\max} - x_{\min}) + x_{\min} \,\mathrm{d}t} \tag{4.23}$$

$$L(F) = \frac{F^2(r-1) + 2F}{r+1}, \tag{4.24}$$

where $r = x_{\max}/x_{\min}$. Thus, we need to find the F value such that $L(F) = 0.5$ and F can be evaluated as

$$L(F) = \frac{F^2(r-1) + 2F}{r+1} = 0.5 \tag{4.25}$$

$$F = \frac{-2 + \sqrt{2r^2+2}}{2(r-1)} \tag{4.26}$$

Thus, the expected number of `MaxMin`-based assignments is

$$(1-F) \times N = \left(\frac{2r - \sqrt{2r^2+2}}{2(r-1)}\right) N \tag{4.27}$$

■

Corollary 4.3.1 *For $K = 2$ homogenous processors, if the task weights of a dataset are uniformly distributed between $x_{\min}$ and $x_{\max}$, the expected number of* `MaxMin`*-based assignments is greater than $0.28N$.*

Proof: The function of Eq. 4.27 is a monotonically decreasing function which gets its minimum while $r \to \infty$.

$$\lim_{r\to\infty} \frac{2r - \sqrt{2r^2+2}}{2(r-1)} = \frac{2-\sqrt{2}}{2} \approx 0.28. \tag{4.28}$$

■

According to Theorem 4.3.1, for a skewed dataset with a typical α value of 2.33 [56], the expected upper bound on the number of `MaxMin`-based assignments to be performed by `MaxMin+` is $0.061N$. That is, at most 6.1% of the assignments will be expensive `MaxMin`-based assignments. This approximately corresponds to a speedup of 16 with respect to `MaxMin`.

According to Theorem 4.3.2, for a uniform dataset with $x_{\max}/x_{\min} = 2$, the expected number of `MaxMin`-based assignments to be performed by `MaxMin+` is

41% of the total number of assignments. These theoretical findings show that the relative speedup of `MaxMin+` over `MaxMin` is expected to be much higher on skewed datasets. The experimental results given in Section 4.6.2.1 validate this expectation.

4.4 Suff+

Despite the success of `Suff` in producing high quality solutions [69, 78, 107], its quadratic running time prevents the application of `Suff` to large datasets. To make `Suff` applicable to large datasets, we combine it with `MinMin+`, under a new heuristic referred to as `Suff+`. The main idea behind the `Suff+` heuristic is to perform critical assignment decisions by `Suff` so that the solution quality is not significantly degraded and perform non-critical assignment decisions by the fast `MinMin+` algorithm. With this approach, we expect a considerable decrease in the execution time of `Suff` with a small potential degradation in the solution quality.

In `Suff+`, the criticality of an assignment decision is determined by the effect of a possible `MinMin+` assignment on the makespan. At each assignment iteration, `Suff+` first computes a task-to-processor assignment according to `MinMin+`. The computed assignment is realized only if it does not lead to an increase in the makespan of the previous iteration. If, however, the `MinMin+`-based assignment increases the makespan, the task-to-processor assignment is recomputed according to the `Suff` heuristic.

The algorithm for `Suff+` is provided in Algorithm 4.13. As in `MaxMin+`, the `MinMin+Init` function (line 3) performs the necessary initializations. Line 5 computes the assignment according to `MinMin+`. The comparison operation at line 6 checks whether makespan will change if the computed assignment is used. Line 7 computes the task-to-processor assignment according to `Suff`.

Algorithm 4.13 SUFF+(x, K, N)

1: $U \leftarrow \{1, 2, \ldots, N\}$
2: $makespan \leftarrow 0$
3: $\langle e, F, Q \rangle \leftarrow$ MINMIN+INIT(x, K)
4: **while** U is not empty **do**
5: $\quad\langle i', k' \rangle \leftarrow$ MINMIN+SELECT(Q, e, K)
6: $\quad$**if** $e_{k'} + x_{i',k'} > makespan$ **then**
7: $\quad\quad\langle i', k' \rangle \leftarrow$ SUFFSELECT(U, e, x, K)
8: $\quad\quad makespan \leftarrow e_{k'} + x_{i',k'}$
9: $\quad A[i'] \leftarrow k'$
10: $\quad e_{k'} \leftarrow e_{k'} + x_{i',k'}$
11: $\quad U \leftarrow U - \{i'\}$
12: $\quad F[i'] \leftarrow$ TRUE
13: $\quad$**for** $k \leftarrow 1$ **to** K **do**
14: $\quad\quad$DELETE(Q_k, i')
15: **return** A

4.5 GA+

Traditionally, the `MinMin` heuristic is used as a submodule in more complex task assignment algorithms. As mentioned in Section 4.1, `GA` is such an algorithm since it uses `MinMin` to find an initial solution. In the literature, `GA` is reported as a slow algorithm, compared to $O(KN^2)$ algorithms such as `MaxMin` and `RC` [10,107].

Herein, we consider `GA` to illustrate the impact of using `MinMin+` instead of `MinMin` on the performance of complex task assignment algorithms. Incorporation of the `MinMin+` heuristic into `GA` leads to an asymptotically faster algorithm, which we refer to as `GA+`. This combination retains the original solution quality of `GA`. `GA+` runs in $O(KN \log N + HNG + HG^2)$ time, making it run much faster than $O(KN^2)$ algorithms and rendering it practical even for large datasets.

Table 4.2: Properties of the datasets

Dataset	N	Task weights Max.	Avg.	α
Social networks				
coauthorship	725,344	672	6.81	3.43 ± 0.04
commonJob	241,233	10,270	7.08	2.30 ± 0.01
Distributed web crawling				
ClueWeb-B	799,115	6.1×10^6	61.56	2.23 ± 0.00
ClueWeb-A	2,483,726	1.5×10^9	1010.50	2.16 ± 0.00
Image-space-parallel direct volume rendering (DVR)				
blunt	20,611	171	90.95	6.51 ± 0.29
comb	32,238	149	64.58	3.84 ± 0.22
Row-parallel sparse matrix vector multiplication (SpMxV)				
barrier2-1	113,076	7,031	33.65	3.78 ± 0.20
language	399,130	11,555	3.05	2.59 ± 0.01
k3plates	11,107	58	34.12	6.42 ± 0.92
big	13,209	12	6.92	7.42 ± 1.57
olafu	16,146	89	62.87	6.29 ± 0.81
mark3jac060	27,449	44	6.22	3.06 ± 0.16
Zhao1	33,861	6	4.92	4.60 ± 2.31
dawson5	51,537	33	19.61	3.02 ± 0.63
epb3	84,617	6	5.48	1.79 ± 0.79
lung2	109,460	8	4.50	2.33 ± 0.26
hood	220,542	77	48.83	6.56 ± 2.16
Lin	256,000	7	6.90	1.16 ± 0.10
pre2	659,033	628	9.04	2.50 ± 0.07

(*) Rows in gray indicate skewed datasets.

4.6 Experimental Results

4.6.1 Datasets

The datasets used in the experiments belong to different application areas: social-network analysis, distributed web crawling, image-space-parallel direct volume rendering (DVR), and row-parallel sparse matrix vector multiplication (SpMxV). In these contexts, the independent task assignment problem arises in load balancing of parallel/distributed applications. These datasets are displayed in Table 4.2.

Our social network datasets (`coauthorship` and `commonJob`) are in the form of sparse graphs. In `coauthorship`, each vertex represents an author and an edge represents the coauthorship relation between two authors. In `commonJob`, each vertex represents an employee and there is an edge between two vertices if the respective employees have ever worked in the same company. The `coauthorship` and `commonJob` datasets are obtained from DBLP[1] and LinkedIn[2], respectively. In both of these graphs, a vertex represents a task to be processed. The degree of a vertex corresponds to the cost of executing the task.

In distributed web crawling datasets (`ClueWeb-A` and `ClueWeb-B`), the tasks represent the web sites and the processors represent the crawlers that will download the pages in the web sites. The weight of a task is set to the number of pages in the respective web site. The `ClueWeb-A` and `ClueWeb-B` datasets, which are obtained from the ClueWeb-09 collection [31], are the largest two datasets among our datasets.

In row-parallel DVR datasets (`blunt` and `comb`), rendering each rectangular pixel block of an image forms a separate task. The weight of a task is set to the expected number of ray-face intersections to be performed while rendering the pixels in the respective pixel block [72]. `blunt` (blunt fin) and `comb` (combustion) are two curvilinear datasets obtained from the NASA Ames Research Center [38].

[1] `http://www.informatik.uni-trier.de/~ley/db/`
[2] `http://www.linkedin.com/`

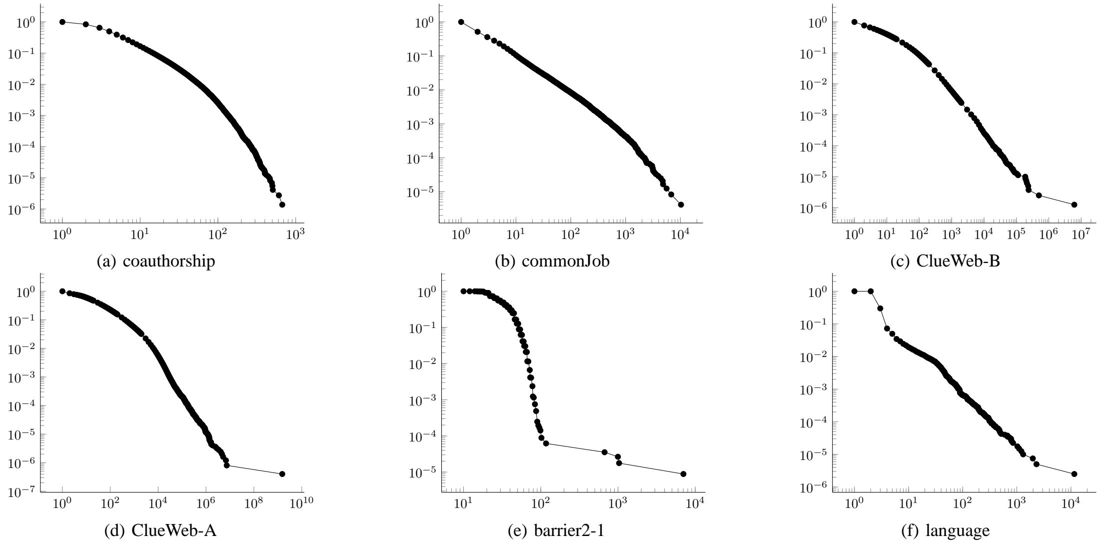


Figure 4.1: Log-log plots of the cumulative density distribution of task weights for skewed datasets. x-axis: weights of tasks, y-axis: cumulative density distribution, i.e., $P(X \geq x)$.

In row-parallel SpMxV datasets, each task corresponds to computing the inner product of a distinct row of the sparse matrix with a dense column vector. The weight of a task is equal to the number of nonzeros in the respective row. We use 13 sparse matrices that are selected from the University of Florida sparse matrix collection [36].

For the distributed web crawling datasets, the ETC value of each task on each crawler is calculated using the techniques described in [19]. For the other datasets, the ETC matrices are constructed using the high machine heterogeneity method discussed in [1]. For each $x_{i,k}$, we multiply the weight of the corresponding task with a random integer in the range $[1 \ldots R]$, where R is the machine heterogeneity constant. Following [1], we selected R as 100 to reflect high machine heterogeneity. For all datasets, the ETC matrices are generated for $K \in \{4, 8, 16, 24, 32\}$ processors. Each dataset and K value combination forms a different assignment instance for our experiments. Since we have 19 datasets and five different K values, we have a total of 95 assignment instances.

In Table 4.2, the Max and Avg columns display the maximum and average

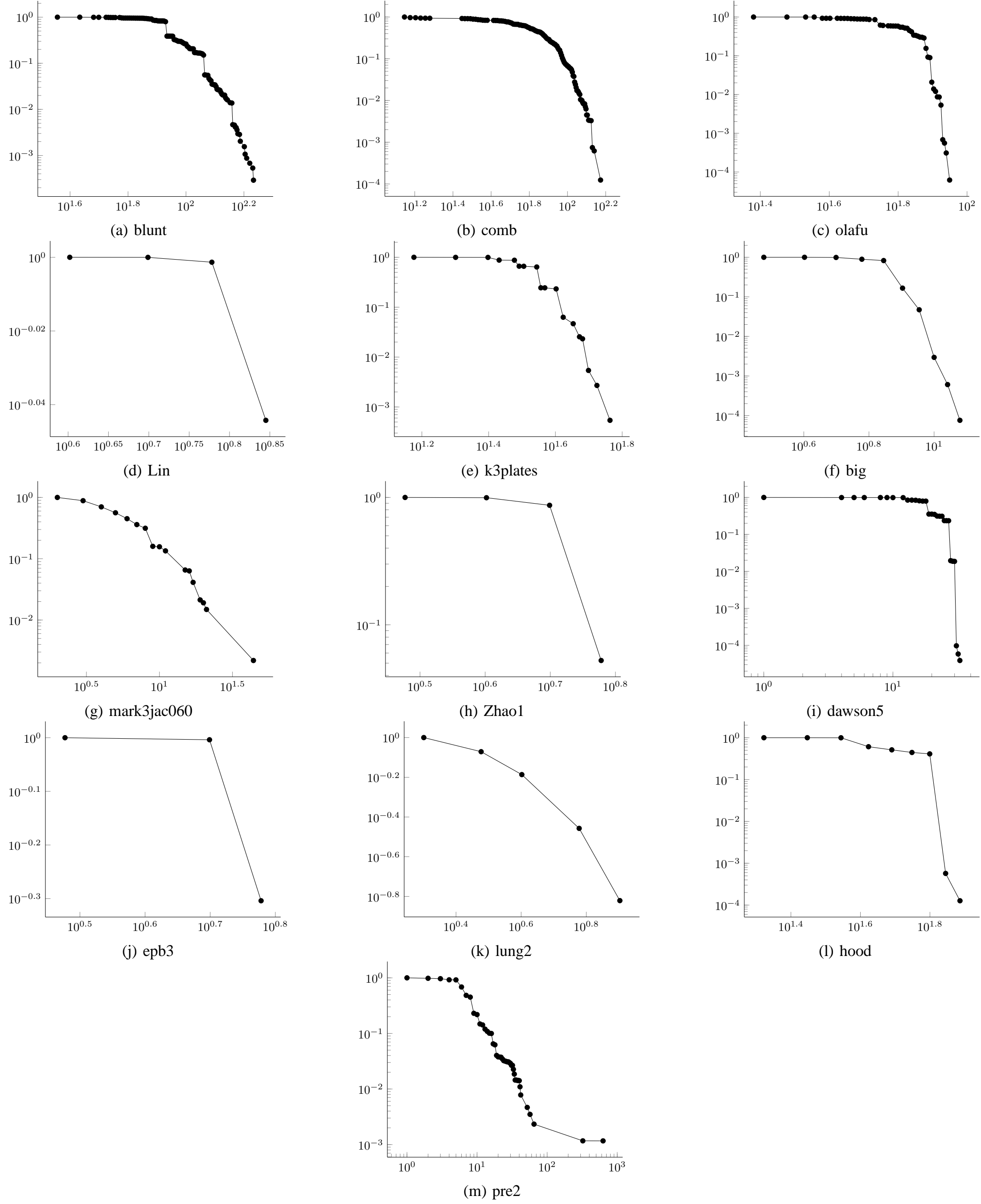


Figure 4.2: Log-log plots of the cumulative density distribution of task weights for non-skewed datasets. x-axis: weights of tasks, y-axis: cumulative density distribution, i.e., $P(X \geq x)$.

task weights, respectively. The α column shows the exponent constant of the power-law distribution $p(w) = Cw^{-\alpha}$ of task weights, together with their error margins. The α values are computed by using the linear least squares method on log-log distributions of the datasets and are used here to identify the datasets with power-law distributions. The datasets that have α values with low error margin and high max/avg ratio are good candidates to have power-law distributions. In this respect, `coauthorship`, `commonJob`, `ClueWeb-B`, `ClueWeb-A`, `barrier2-1`, and `language` datasets are considered to have a power-law distribution. In the remaining tables, the rows are colored in gray to indicate skewed datasets.

Fig. 4.1 displays the log-log plots of the cumulative density distribution of task weights for the skewed datasets. Fig. 4.2 displays the log-log plots of the cumulative density distribution of task weights for the skewed datasets.

4.6.2 Performance Analysis

All of the algorithms are implemented in Java programming language. All experiments were carried out on a Linux workstation equipped with six 2100-MHz quad-core CPUs and 132 GB of memory.

The load balancing quality of the assignment algorithms are compared according to the percent load imbalance ratio defined as

$$\%\mathrm{LI} = 100 \times \frac{M - M^*}{M^*}, \tag{4.29}$$

where M denotes the makespan of an assignment produced by an algorithm and M^* denotes the ideal makespan for the given assignment instance. M^* is computed as

$$M^* = \frac{W^*_{\mathrm{tot}}}{K} = \frac{\sum_i \min_k \{x_{i,k}\}}{K}, \tag{4.30}$$

where W^*_{tot} is the execution time obtained when the tasks are assigned to their favorite processor. This value forms a rather loose lower bound for the makespan. The optimal makespan is potentially greater than M^*.

Table 4.3: Percent load imbalance values for social network datasets

		Original heuristics				Proposed heuristics			
						MM			GA
Dataset	K	MxM	Suff	RC	RASA	MM+	MxM+	Suff+	GA+
coauthorship	4	204.94	0.08	**0.01**	163.35	0.10	0.13	0.02	0.04
	8	280.50	0.58	**0.02**	229.68	0.13	0.07	0.06	0.07
	16	316.25	1.86	**0.10**	263.86	0.48	0.11	0.24	0.30
	24	315.95	2.43	0.22	266.97	0.76	**0.11**	0.34	0.43
	32	310.82	2.66	**0.19**	262.12	1.69	0.30	0.80	0.96
commonJob	4	163.19	0.71	1.07	143.40	1.87	0.72	**0.53**	0.81
	8	218.67	2.51	**0.63**	192.47	8.97	1.46	1.86	3.75
	16	239.99	5.26	**3.28**	212.25	18.92	3.87	10.14	9.24
	24	235.61	5.62	**5.08**	213.56	23.90	8.77	17.85	14.50
	32	227.71	6.97	**4.58**	204.58	37.28	14.51	16.81	23.42

Table 4.4: Percent load imbalance values for distributed web crawling datasets

		Original heuristics				Proposed heuristics			
						MM			GA
Dataset	K	MxM	Suff	RC	RASA	MM+	MxM+	Suff+	GA+
ClueWeb-B	4	81.49	22.28	19.91	80.06	41.82	**17.24**	18.52	37.62
	8	175.88	102.35	103.61	173.05	168.72	99.63	**99.02**	159.99
	16	230.77	162.19	161.96	227.42	319.82	160.48	**155.16**	306.18
	24	286.10	**222.08**	224.38	282.29	476.91	230.77	230.77	458.82
	32	**323.97**	**323.97**	324.50	**323.97**	607.80	**323.97**	**323.97**	589.19
ClueWeb-A	4	**172.02**	**172.02**	173.35	**172.02**	205.18	**172.02**	**172.02**	204.82
	8	**436.41**	**436.41**	436.85	**436.41**	482.96	**436.41**	**436.41**	482.31
	16	**802.88**	**802.88**	802.92	**802.88**	891.27	**802.88**	**802.88**	889.61
	24	**1286.95**	**1286.95**	1286.98	**1286.95**	1393.57	**1286.95**	**1286.95**	1388.59
	32	**1763.49**	**1763.49**	1763.53	**1763.49**	1868.91	**1763.49**	**1763.49**	1862.57

Table 4.5: Percent load imbalance values for parallel DVR datasets

		Original heuristics				Proposed heuristics			
						MM			GA
Dataset	K	MxM	Suff	RC	RASA	MM+	MxM+	Suff+	GA+
`blunt`	4	185.24	0.10	0.06	102.04	0.11	1.12	0.07	**0.04**
	8	253.94	0.65	0.27	155.03	0.29	0.65	0.23	**0.12**
	16	276.43	1.86	0.52	175.31	0.60	0.60	0.54	**0.34**
	24	275.48	2.25	1.37	176.10	1.02	0.78	1.02	**0.47**
	32	269.72	2.52	2.07	172.12	1.42	1.07	1.18	**0.74**
`comb`	4	187.83	0.10	0.05	116.36	0.08	0.67	0.09	**0.03**
	8	252.82	0.74	0.12	169.19	0.16	0.48	0.17	**0.08**
	16	278.85	1.83	0.43	195.78	0.49	0.31	0.35	**0.24**
	24	276.05	2.56	0.86	191.02	0.85	0.66	0.83	**0.47**
	32	271.01	2.81	1.40	189.24	0.94	0.70	0.92	**0.55**

Table 4.6: Percent load imbalance values for parallel SpMxV datasets

Dataset	K	Original heuristics				Proposed heuristics			
		MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
barrier2-1	4	202.22	0.12	**0.01**	119.77	0.89	0.46	0.04	0.15
	8	278.87	0.59	**0.03**	180.31	2.25	0.26	0.09	0.51
	16	310.18	1.38	**0.09**	208.49	0.36	0.19	0.12	0.18
	24	311.27	2.10	0.30	210.86	7.56	0.28	**0.21**	2.42
	32	303.48	2.25	0.30	207.41	1.39	**0.26**	0.30	0.77
language	4	198.73	0.38	0.03	121.62	1.68	0.27	**0.03**	0.63
	8	286.72	2.59	0.33	186.64	7.30	**0.27**	0.44	3.23
	16	315.71	3.98	1.31	214.60	27.98	**1.15**	2.01	22.87
	24	319.59	2.51	**0.57**	219.26	5.72	0.57	4.49	3.20
	32	308.00	4.37	**1.79**	212.24	58.74	4.49	3.70	51.44
olafu	4	184.48	0.16	0.11	104.11	0.09	1.04	0.11	**0.04**
	8	247.81	0.80	0.34	152.60	0.27	0.58	0.28	**0.12**
	16	269.75	1.78	0.88	172.10	0.81	0.69	0.63	**0.37**
	24	267.79	2.79	1.47	172.49	0.96	0.90	1.22	**0.57**
	32	258.30	2.98	2.30	171.92	1.14	1.15	1.23	**0.76**
Lin	4	218.44	0.01	**0.01**	115.61	**0.01**	0.41	0.01	**0.01**
	8	324.60	0.13	0.01	193.65	0.01	0.29	0.03	**0.01**
	16	361.38	0.62	0.05	223.68	0.05	0.17	0.05	**0.02**
	24	358.59	1.01	0.07	223.51	0.07	0.14	0.06	**0.04**
	32	349.33	1.12	0.09	219.01	0.10	0.13	0.09	**0.09**
k3plates	4	179.00	0.17	0.07	101.07	0.10	1.30	0.16	**0.04**
	8	241.49	1.06	0.38	146.46	0.32	0.87	0.43	**0.18**
	16	260.96	2.16	1.21	166.24	1.01	0.75	0.96	**0.52**
	24	255.10	2.82	1.75	165.63	1.70	1.19	1.16	**0.76**
	32	249.78	3.23	2.54	161.44	3.29	1.78	2.73	**1.36**
big	4	181.42	0.12	**0.03**	100.12	0.11	1.21	0.11	0.04
	8	248.43	0.86	0.31	152.67	0.23	0.81	0.27	**0.11**
	16	268.72	2.06	0.82	169.92	0.83	0.95	0.79	**0.45**
	24	264.66	2.60	1.59	169.22	1.22	1.11	0.87	**0.79**
	32	254.69	3.23	2.62	167.07	2.75	**1.45**	2.24	1.62
mark3jac060	4	176.46	0.35	**0.09**	118.32	0.25	0.65	0.09	0.17
	8	234.30	1.17	**0.16**	165.75	0.57	0.43	0.24	0.21
	16	260.27	2.76	1.02	189.26	1.95	**0.64**	0.89	0.82
	24	257.10	3.61	3.91	187.85	4.45	**1.39**	1.60	2.20
	32	248.52	3.40	**1.32**	182.51	2.13	1.40	2.69	1.47

Table 4.7: Percent load imbalance values for parallel SpMxV datasets (2)

		Original heuristics				Proposed heuristics			
Dataset	K	MxM	Suff	RC	RASA	MM MM+	MxM+	Suff+	GA GA+
`Zhao1`	4	196.65	0.05	0.03	107.07	0.03	0.94	0.04	**0.01**
	8	272.92	0.47	0.14	163.69	0.10	0.53	0.16	**0.05**
	16	302.54	1.37	0.38	187.93	0.30	0.45	0.34	**0.14**
	24	296.84	1.80	0.54	188.41	0.54	0.47	0.39	**0.33**
	32	287.62	2.21	0.85	181.98	1.01	**0.54**	0.73	0.56
`dawson5`	4	196.91	0.07	0.04	111.86	0.03	0.70	0.03	**0.02**
	8	268.83	0.49	0.08	168.32	0.09	0.34	0.09	**0.04**
	16	296.58	1.50	0.36	194.61	0.29	0.35	0.20	**0.14**
	24	293.83	1.97	0.57	192.20	0.42	0.30	0.29	**0.24**
	32	291.60	2.12	0.54	192.71	0.56	0.47	0.52	**0.38**
`epb3`	4	208.07	0.02	0.01	112.18	0.01	0.60	0.02	**0.01**
	8	292.89	0.28	0.05	175.48	0.04	0.36	0.07	**0.02**
	16	325.47	1.06	0.14	203.47	0.14	0.27	0.13	**0.08**
	24	320.46	1.40	0.24	201.55	0.24	0.23	0.18	**0.13**
	32	313.97	1.53	0.33	198.12	0.38	0.29	0.34	**0.22**
`lung2`	4	201.66	0.04	0.01	124.79	0.03	0.43	0.02	**0.01**
	8	278.07	0.46	0.05	184.74	0.07	0.26	0.09	**0.04**
	16	308.15	1.46	0.18	210.33	0.13	0.21	0.18	**0.08**
	24	308.12	1.91	0.33	212.92	0.27	0.28	0.19	**0.17**
	32	299.94	2.01	0.46	208.58	0.55	**0.28**	0.42	0.32
`hood`	4	215.54	0.01	**0.01**	121.10	**0.01**	0.41	**0.01**	**0.01**
	8	305.25	0.21	0.03	189.84	0.02	0.20	0.04	**0.01**
	16	340.15	0.93	0.07	219.67	0.08	0.16	0.05	**0.05**
	24	340.49	1.29	0.14	221.24	0.10	0.13	0.12	**0.06**
	32	332.24	1.45	0.16	216.91	0.13	0.15	0.16	**0.09**
`pre2`	4	211.05	0.07	**0.01**	137.48	0.13	0.20	**0.01**	0.06
	8	294.44	0.50	0.17	204.69	0.37	**0.07**	0.20	0.16
	16	329.40	1.63	0.70	236.59	0.92	**0.39**	0.67	0.63
	24	329.62	2.08	2.05	237.63	1.67	**0.72**	1.59	1.04
	32	323.29	2.42	**0.39**	234.68	1.49	0.87	1.61	1.22

Table 4.8: Averages of percent load imbalance values over all datasets

Dataset	K	Original heuristics MxM	Suff	RC	RASA	Proposed heuristics MM MM+	MxM+	Suff+	GA GA+
Skewed	4	170.43	32.60	32.40	133.37	41.92	31.81	31.86	40.68
	8	279.51	90.84	90.25	233.09	111.72	89.68	89.65	108.31
	16	369.30	162.92	161.61	321.58	209.80	161.45	161.76	204.73
	24	459.25	253.61	252.92	413.32	318.07	254.57	256.77	311.33
	32	539.58	350.62	349.15	495.64	429.30	351.17	351.51	421.39
Non-skewed	4	195.60	0.10	0.04	113.24	0.08	0.74	0.06	0.04
	8	270.44	0.60	0.16	170.93	0.20	0.45	0.18	0.09
	16	298.36	1.62	0.52	195.76	0.58	0.46	0.44	0.30
	24	295.70	2.16	1.15	195.37	1.04	0.64	0.73	0.56
	32	288.46	2.39	1.16	192.02	1.22	0.79	1.14	0.72

Tables 4.3–4.7 display the load imbalance values for 4-, 8-, 16-, 24-, and 32-way assignments obtained by the existing (baseline) and proposed heuristics for different types of datasets. Table 4.8 displays load imbalance averages for different K values over all datasets. In these tables, we display the results of `MinMin` and `MinMin+` in the same column, since these heuristics attain the same results. The results of `GA` and `GA+` are displayed in the same column due to the same reason.

Tables 4.9–4.13 display the running times of the heuristics for different types of datasets. Table 4.14 displays running time averages for different K values over all datasets. These averages are obtained by normalizing the running time values with those attained by the `MinMin+` heuristic.

In Tables 4.6, Tables 4.7, 4.12 and 4.13 the performance results for row-parallel SpMxV datasets are presented by splitting data into two tables. The average performance results displayed in Tables 4.8 and 4.14 are computed by considering the performance results of all datasets.

In Tables 4.3–4.7, the bold value(s) in each row indicate the best solution(s) in terms of load balancing performance for the respective assignment instance. In all tables, the `MinMin`, `MinMin+`, `MaxMin`, and `MaxMin+` heuristics are abbreviated as MM, MM+, MxM, and MxM+, respectively.

Table 4.9: Running times (seconds) of heuristics for social network datasets

Dataset	K	Original heuristics MM	MxM	Suff	RC	RASA	GA	Proposed heuristics MM+	MxM+	Suff+	GA+
coauthorship	4	53,859.2	63,053.1	67,678.7	89,023.9	64,896.3	54,884.8	5.7	172.5	387.4	1,031.2
	8	70,204.6	66,434.0	97,158.5	1.2×10^5	81,245.7	72,146.3	11.5	71.8	218.3	1,953.2
	16	1.2×10^5	1.4×10^5	2.0×10^5	1.9×10^5	1.4×10^5	1.3×10^5	20.9	66.3	168.2	4,407.1
	24	1.8×10^5	2.0×10^5	2.8×10^5	2.4×10^5	1.7×10^5	1.9×10^5	33.4	85.7	235.6	4,277.8
	32	2.1×10^5	2.1×10^5	3.5×10^5	2.8×10^5	1.9×10^5	2.2×10^5	40.9	84.8	171.4	4,414.9
commonJob	4	8,276.9	6,810.9	5,346.2	4,883.6	7,059.9	8,781.3	1.3	2.3	2.7	505.6
	8	8,242.8	9,522.5	9,031.0	9,810.2	8,604.5	9,375.7	2.4	3.0	3.7	1,135.3
	16	13,506.1	13,627.6	12,932.8	13,657.7	13,847.8	14,905.9	2.6	5.9	4.2	1,402.5
	24	18,835.1	18,537.7	20,593.0	26,190.4	17,578.4	20,346.4	7.7	9.7	9.6	1,519.0
	32	24,104.4	37,576.2	26,927.1	26,379.2	21,281.3	25,619.6	9.7	10.5	9.2	1,524.9

Table 4.10: Running times (seconds) of heuristics for distributed web crawling datasets

Dataset	K	Original heuristics MM	MxM	Suff	RC	RASA	GA	Proposed heuristics MM+	MxM+	Suff+	GA+
ClueWeb-B	4	73,814.2	75,260.0	78,577.7	1.1×10^5	90,773.2	77,284.4	4.1	5.3	7.7	3,474.3
	8	1.2×10^5	88,850.7	79,415.0	1.4×10^5	1.1×10^5	1.2×10^5	9.5	11.5	13.6	4,386.9
	16	2.3×10^5	1.4×10^5	1.9×10^5	2.8×10^5	1.3×10^5	2.4×10^5	18.2	17.6	22.5	5,144.0
	24	2.9×10^5	2.6×10^5	2.9×10^5	3.7×10^5	1.8×10^5	2.9×10^5	36.5	42.4	28.2	4,059.6
	32	4.1×10^5	3.2×10^5	3.6×10^5	4.3×10^5	2.2×10^5	4.1×10^5	47.3	41.6	46.0	4,169.8
ClueWeb-A	4	6.7×10^5	8.1×10^5	7.3×10^5	9.3×10^5	6.5×10^5	6.9×10^5	19.6	19.4	20.8	12,573.3
	8	8.4×10^5	1.1×10^6	1.0×10^6	1.4×10^6	7.9×10^5	8.5×10^5	39.2	38.8	51.1	11,473.6
	16	1.9×10^6	1.7×10^6	1.8×10^6	2.8×10^6	1.2×10^6	2.0×10^6	60.5	84.2	89.7	12,936.7
	24	2.7×10^6	2.6×10^6	3.0×10^6	2.9×10^6	1.8×10^6	2.7×10^6	106.2	112.3	141.0	14,059.2
	32	3.3×10^6	2.9×10^6	3.2×10^6	3.5×10^6	2.8×10^6	3.4×10^6	183.9	174.5	193.1	14,231.3

4.6.2.1 Comparison with Traditional Counterparts

In this subsection, we discuss the performance of each proposed heuristic against its traditional counterpart.

`MinMin+` versus `MinMin`: As mentioned in Section 4.2, `MinMin+` finds exactly the same solutions as `MinMin`. However, `MinMin+` is several orders of magnitude faster than `MinMin` in all assignment instances. On average, `MinMin+` is 5603-, 3703-, 4192-, 3214-, and 2947-times faster than `MinMin` in 4-, 8-, 16-, 24-, and 32-way assignments, respectively.

As expected, the speedup of `MinMin+` over `MinMin` increases with increasing number of tasks. For the 16-way assignment of the largest dataset `ClueWeb-A`, which contains about 2.5 million tasks, `MinMin` finds a solution in about 22 days while `MinMin+` finds the same solution in about a minute, i.e., `MinMin+` runs about 31,400 times faster than `MinMin`.

Table 4.11: Running times (seconds) of heuristics for parallel DVR datasets

Dataset	K	Original heuristics						Proposed heuristics			
		MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
blunt	4	15.3	15.1	17.8	21.7	18.9	25.6	0.0	0.7	2.5	10.3
	8	26.0	23.9	34.0	43.1	29.9	40.7	0.1	0.5	2.1	14.7
	16	69.2	59.7	100.8	107.8	132.4	90.7	0.2	0.4	1.8	21.7
	24	208.5	228.7	163.1	174.5	164.8	234.4	0.3	0.8	4.3	26.2
	32	259.2	287.1	334.6	246.1	231.6	291.8	0.3	0.8	3.8	32.9
comb	4	56.3	39.1	47.0	188.5	85.8	70.2	0.1	1.4	5.0	14.0
	8	88.0	124.3	93.5	113.0	186.7	114.7	0.2	0.8	3.9	26.8
	16	159.5	191.2	279.7	236.9	256.7	200.6	0.3	1.0	3.8	41.4
	24	314.0	289.2	356.3	466.2	350.3	360.7	0.6	1.7	6.7	47.3
	32	437.3	445.9	446.2	457.5	436.5	475.7	0.6	1.5	6.8	38.9

MaxMin+ versus MaxMin: MaxMin+ finds drastically better solutions than MaxMin in all assignment instances, except for the 32-way assignment of ClueWeb-B and the assignment instances of ClueWeb-A, where both heuristics find solutions with the same makespan. The averages displayed in Table 4.8 demonstrate the large quality difference between MaxMin+ and MaxMin. On average, MaxMin+ attains average load imbalance values of 177.74% and 0.62% compared to 363.61% and 269.71% of MaxMin, for skewed and non-skewed datasets, respectively. Moreover, MaxMin+ is several orders of magnitude faster than MaxMin in all assignment instances. On average, MaxMin+ runs 6917- and 404-times faster than MaxMin for skewed and non-skewed datasets, respectively. Note that the performance gaps between MaxMin+ and MaxMin in load balancing and running time are much higher in non-skewed datasets compared to skewed datasets in favor of MaxMin+. The former is expected since MaxMin is highly tuned for skewed datasets and fails to find good solutions for non-skewed datasets, whereas MaxMin+ is a more balanced heuristic. The latter is also expected since skewed datasets generally contain much larger number of tasks than non-skewed datasets.

Tables 4.15 and 4.16 display the number of MaxMin-based assignments performed by MaxMin+. As seen in these tables, in general, the number of MaxMin-based assignments considerably decreases with increasing K values, thus conforming with the expectation given in Section 4.3. This behavior explains the decrease in the running time performance gap between MaxMin+ and MinMin+ with increasing K as shown in Table 4.14. Even for the smallest K value of

Table 4.12: Running times (seconds) of heuristics for parallel SpMxV datasets

Dataset	K	Original heuristics MM	MxM	Suff	RC	RASA	GA	Proposed heuristics MM+	MxM+	Suff+	GA+
barrier2-1	4	1,044.0	1,245.0	1,978.2	1,065.2	1,505.8	1,176.6	0.6	20.6	74.8	133.1
	8	1,809.8	1,835.4	2,295.3	2,343.4	2,008.1	2,134.5	1.1	15.9	61.9	325.8
	16	3,356.1	3,138.0	3,961.0	4,697.8	3,254.9	3,636.5	2.2	20.4	62.2	282.7
	24	4,534.0	4,893.5	5,511.7	5,959.1	4,099.6	5,150.5	3.7	15.1	73.9	620.3
	32	5,078.4	5,810.1	6,360.4	6,551.5	6,345.0	5,418.7	3.5	14.1	83.4	343.9
language	4	15,081.9	16,432.9	16,562.0	26,088.5	25,826.6	16,413.0	2.3	65.9	166.8	1,333.3
	8	25,924.4	23,470.0	24,928.3	34,444.6	34,882.4	28,029.7	4.7	15.9	40.0	2,110.1
	16	39,780.2	34,559.5	47,030.2	71,420.6	51,280.7	42,259.3	11.5	12.6	12.0	2,490.6
	24	74,398.5	76,276.2	75,045.1	90,750.8	70,951.8	77,000.6	22.4	32.3	50.2	2,624.6
	32	71,323.0	67,528.2	71,835.9	77,366.0	77,218.7	73,990.1	18.2	21.9	15.9	2,685.3
olafu	4	9.1	9.8	11.0	13.4	13.2	15.7	0.0	0.4	1.5	6.7
	8	14.0	13.6	49.8	22.0	18.9	22.6	0.1	0.3	1.2	8.7
	16	35.8	40.7	38.5	58.1	54.3	54.9	0.2	0.3	1.0	19.3
	24	81.9	129.2	97.7	106.2	84.4	101.7	0.3	0.4	2.4	20.1
	32	180.4	156.1	136.8	143.1	131.6	196.1	0.3	0.5	2.1	15.9
Lin	4	6,987.9	8,185.9	7,401.1	10,191.5	7,977.8	7,234.2	1.2	200.8	684.1	247.5
	8	8,309.2	10,809.7	11,219.6	16,430.8	10,947.9	8,688.3	2.0	129.2	556.3	381.1
	16	15,901.8	24,876.3	20,575.5	28,687.5	16,427.6	16,374.7	6.0	117.8	784.2	478.9
	24	23,305.1	23,062.5	25,086.5	31,325.5	23,233.7	23,847.7	8.5	109.9	794.6	551.1
	32	28,725.5	29,544.6	31,139.2	45,157.7	29,805.8	29,184.5	10.5	119.4	766.7	469.5
k3plates	4	3.6	5.5	5.1	5.0	5.8	7.3	0.0	0.2	0.7	3.7
	8	5.3	9.4	8.1	7.3	10.5	10.8	0.0	0.1	0.6	5.6
	16	22.6	27.6	15.4	23.0	13.2	32.2	0.1	0.9	0.5	9.7
	24	30.0	34.9	64.2	63.9	29.1	39.7	0.4	0.5	0.9	10.1
	32	30.5	28.1	50.9	53.3	24.2	37.6	1.5	0.2	0.8	8.7
big	4	5.9	8.9	9.3	8.1	7.7	10.4	0.0	0.5	1.1	4.5
	8	8.6	9.3	11.9	13.6	11.2	16.2	0.1	0.2	0.9	7.6
	16	19.7	17.6	26.2	39.6	22.5	28.6	0.1	0.2	0.7	9.0
	24	28.1	74.1	68.1	121.5	47.9	37.7	0.2	0.3	1.3	9.8
	32	44.0	44.2	74.8	79.1	57.3	50.2	0.2	0.4	1.2	6.4
mark3jac060	4	47.9	56.2	67.9	40.5	79.8	59.5	0.1	0.7	2.2	11.7
	8	79.5	86.4	87.4	107.9	153.6	93.5	0.2	0.4	1.6	14.1
	16	161.2	165.0	159.6	142.9	118.8	180.6	0.3	0.5	2.3	19.6
	24	194.6	412.9	405.6	302.2	228.0	224.5	0.6	0.9	2.2	30.5
	32	207.2	297.9	332.0	224.0	325.9	230.8	0.6	0.9	3.7	24.2

four, the number of MaxMin-based assignments is much smaller than the number of MinMin-based assignments for each instance. For $K = 4$, the worst case occurs for the big matrix, where only 9.25% of the assignments are MaxMin-based assignments. These results show that the expected number of MaxMin-based assignments given in Theorem 4.3.1 for $K = 2$ homogenous processors is a rather loose upper bound for $K \geq 4$ heterogeneous processors.

As seen in Table 4.15, MaxMin+ makes only one MaxMin-based assignment for the 32-way assignment of ClueWeb-B and all K-way assignments of ClueWeb-A. ClueWeb-A has an extremely large task whose weight is greater than the sum

Table 4.13: Running times (seconds) of heuristics for parallel SpMxV datasets (2)

Dataset	K	Original heuristics						Proposed heuristics			
		MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
	4	80.7	57.9	92.8	73.9	72.8	93.6	0.2	2.7	7.7	13.0
	8	109.8	117.8	162.0	154.7	131.8	130.7	0.2	2.6	6.7	21.2
Zhao1	16	279.3	270.5	311.1	288.0	234.0	306.1	0.3	1.1	7.1	27.1
	24	268.0	510.0	461.3	489.1	278.6	291.4	0.6	1.8	8.5	24.0
	32	487.0	444.3	443.8	346.6	384.8	509.6	0.6	2.7	10.1	23.3
	4	219.5	327.8	324.7	324.8	271.6	245.0	0.9	4.4	14.7	26.4
	8	562.8	502.0	409.2	530.7	406.9	602.8	0.3	2.3	14.6	40.4
dawson5	16	598.8	705.8	866.7	538.0	687.2	678.4	0.5	2.4	17.3	80.1
	24	717.6	1,547.7	1,090.4	1,261.0	822.6	769.7	1.1	4.2	13.7	53.2
	32	1,060.4	935.9	1,151.2	1,167.0	1,045.1	1,130.8	1.1	3.5	22.1	71.4
	4	1,122.1	866.2	760.6	706.1	675.2	1,172.0	0.3	12.4	54.3	50.2
	8	783.0	1,091.2	1,047.0	1,740.2	1,049.5	846.0	0.5	7.9	50.0	63.5
epb3	16	1,894.5	1,636.9	1,975.4	1,533.6	1,706.4	2,016.7	0.9	6.9	52.7	123.1
	24	2,235.8	1,811.0	3,034.0	2,927.4	2,400.4	2,355.4	1.8	11.7	59.6	121.4
	32	2,273.5	2,963.6	2,931.3	2,959.6	2,702.3	2,387.2	2.2	12.0	70.3	115.9
	4	1,349.1	1,493.3	1,504.6	1,018.8	1,413.5	1,404.6	0.4	17.6	69.3	56.0
	8	1,860.0	2,031.7	2,199.8	2,509.3	1,843.3	1,999.3	0.9	12.3	58.0	140.2
lung2	16	3,214.7	2,971.9	3,804.1	2,868.1	2,519.5	3,378.4	1.7	11.3	51.5	165.4
	24	4,261.0	5,690.3	6,252.9	5,359.6	4,478.7	4,443.2	3.0	12.0	71.2	185.3
	32	4,349.4	5,287.2	5,703.2	5,630.6	5,064.2	4,543.0	3.5	20.4	96.6	197.1
	4	6,010.0	7,373.3	5,346.4	5,546.9	7,877.5	6,206.7	1.4	109.5	398.2	198.0
	8	7,178.2	6,601.1	7,658.0	9,781.4	9,298.1	7,471.1	2.8	72.3	296.8	295.8
hood	16	12,451.9	16,071.5	14,179.9	14,639.6	14,818.7	12,878.2	5.0	58.3	274.9	431.3
	24	21,433.8	21,570.2	23,065.8	25,256.5	19,730.1	21,955.1	9.0	91.9	494.1	530.2
	32	20,735.2	30,549.0	22,441.7	20,808.9	25,090.0	21,276.0	8.4	68.1	430.9	549.2
	4	44,568.1	56,055.2	62,351.2	81,854.7	50,118.6	45,398.7	5.1	254.0	668.8	835.6
	8	54,970.0	80,027.6	64,028.6	1.0×10^5	65,710.6	57,131.3	10.4	54.3	161.3	2,171.6
pre2	16	99,706.8	91,535.7	1.2×10^5	2.2×10^5	1.1×10^5	1.0×10^5	18.5	33.9	47.8	3,719.3
	24	1.5×10^5	2.3×10^5	2.4×10^5	2.0×10^5	1.3×10^5	1.5×10^5	32.2	41.0	84.8	4,278.1
	32	1.5×10^5	2.6×10^5	2.8×10^5	3.4×10^5	1.7×10^5	1.5×10^5	36.3	47.9	68.8	4,375.2

Table 4.14: Normalized running time averages over all datasets

Dataset	K	Original heuristics						Proposed heuristics			
		MM	MxM	Suff	RC	RASA	GA	MM+	MxM+	Suff+	GA+
	4	12,813.9	14,307.1	13,863.1	17,964.0	14,379.7	13,294.8	1.0	16.7	46.8	481.9
	8	8,357.6	9,069.9	8,878.2	12,122.8	8,653.1	8,711.3	1.0	4.5	14.5	354.7
Skewed	16	10,134.9	8,614.4	9,924.2	14,029.0	7,595.1	10,397.5	1.0	3.0	6.9	263.6
	24	7,604.2	7,363.6	8,610.4	8,867.2	5,623.8	7,745.3	1.0	1.9	5.4	142.1
	32	6,643.6	6,186.7	6,981.4	7,304.4	5,482.1	6,755.3	1.0	1.7	5.3	112.8
	4	2,274.3	2,556.8	2,501.8	2,988.4	2,488.6	2,435.1	1.0	38.0	130.5	161.8
	8	1,555.4	1,902.1	1,858.3	2,553.1	1,907.3	1,703.4	1.0	13.1	59.1	149.0
Non-skewed	16	1,449.6	1,577.0	1,766.1	2,168.1	1,539.7	1,565.2	1.0	5.9	29.5	116.6
	24	1,187.6	1,558.1	1,625.2	1,620.5	1,171.9	1,250.5	1.0	4.1	21.5	63.9
	32	1,241.6	1,618.3	1,639.7	1,814.9	1,343.0	1,298.0	1.0	3.8	20.4	57.4

Table 4.15: Number of MaxMin-based assignments performed by MaxMin+ for social network, distributed web crawling and parallel DVR datasets

Social network			Distributed web crawling			Parallel DVR		
Dataset	K	m	Dataset	K	m	Dataset	K	m
coauthorship (**N=725,344**)			**ClueWeb-B** (**N=799,115**)			**blunt** (**N=20,611**)		
	4	13,528		4	257		4	1,840
	8	3,631		8	289		8	696
	16	1,190		16	9		16	282
	24	686		24	2		24	172
	32	444		32	1		32	128
commonJob (**N=241,233**)			**ClueWeb-A** (**N=2,483,726**)			**comb** (**N=32,238**)		
	4	441		4	1		4	2,466
	8	93		8	1		8	912
	16	23		16	1		16	370
	24	11		24	1		24	226
	32	9		32	1		32	165

of the weights of all other tasks. The assignment of such a large task to its favorite processor avoids the need for a second `MaxMin`-based assignment in future iterations. A similar reasoning holds for the 32-way assignment of `ClueWeb-B`. In fact, `MaxMin` is also expected to find a "good" solution in such assignment instances. As seen in Tables 4.3–4.7, these are the only assignment instances where `MaxMin` was able to find a solution with the same makespan as `MaxMin+`.

`MaxMin+` versus `RASA`: Although `RASA` finds slightly better solutions than `MaxMin`, `MaxMin+` finds significantly better solutions than `RASA` in all assignment instances, except for the 32-way assignment of `ClueWeb-B` and the assignment instances of `ClueWeb-A` , where all three heuristics find solutions with the same makespan. On average, `MaxMin+` attains average load imbalance values of 177.74% and 0.62% compared to 319.40% and 173.46% of `RASA`, for skewed and non-skewed datasets, respectively. These results validate the success of the proposed adaptive selection policy of `MaxMin+` over that of `RASA`. `MaxMin+` is several orders of magnitude faster than `RASA` in all assignment instances. On average, `MaxMin+` runs 5953- and 333-times faster than `RASA` for skewed and non-skewed datasets, respectively.

Table 4.16: Number of MaxMin-based selections performed by MaxMin+ for parallel SpMxV datasets

Dataset	K	m	Dataset	K	m	Dataset	K	m
barrier2-1 (N=113,076)			**language (N=399,130)**			**olafu (N=16,146)**		
	4	8,233		4	8,986		4	1,416
	8	2,987		8	1,093		8	535
	16	1,198		16	114		16	227
	24	668		24	137		24	138
	32	496		32	11		32	103
Lin (N=256,000)			**k3plates (N=11,107)**			**big (N=13,209)**		
	4	23,376		4	1,009		4	1,222
	8	8,882		8	393		8	456
	16	3,666		16	160		16	190
	24	2,246		24	98		24	121
	32	1,634		32	72		32	87
mark3jac060 (N=27,449)			**Zhao1 (N=33,861)**			**dawson5 (N=51,537)**		
	4	1,492		4	3,130		4	4,281
	8	509		8	1,192		8	1,580
	16	186		16	483		16	660
	24	110		24	307		24	395
	32	81		32	221		32	290
epb3 (N=84,617)			**lung2 (N=109,460)**			**hood (N=220,542)**		
	4	7,667		4	7,767		4	18,151
	8	2,936		8	2,800		8	6,786
	16	1,187		16	1,105		16	2,745
	24	750		24	692		24	1,680
	32	541		32	507		32	1,235
pre2 (N=84,617)								
	4	20,184						
	8	3,172						
	16	372						
	24	173						
	32	127						

`Suff+` versus `Suff`: Out of 95 assignment instances, `Suff+` finds better solutions than `Suff` in 83 instances, whereas `Suff` finds better solutions than `Suff+` in only six instances. In the remaining six assignment instances (five assignment instances of `ClueWeb-A` and the 32-way assignment of `ClueWeb-B`), both `Suff` and `Suff+` find solutions with the same makespan. As seen in Table 4.8, in terms of average load balancing quality, `Suff+` shows comparable performance with `Suff` for skewed datasets, whereas `Suff+` performs better than `Suff` for non-skewed datasets. On average, `Suff+` attains average load imbalance values of 178.31% and 0.51% compared to 178.12% and 1.37% of `Suff`, for skewed and non-skewed datasets, respectively. As seen in Table 4.14, `Suff+` is a few orders of magnitude faster than `Suff` in all assignment instances. On average, `Suff+` runs 6078- and 194-times faster than `Suff` for skewed and non-skewed datasets, respectively.

`GA+` versus `GA`: As mentioned in Section 4.5, `GA+` finds exactly the same solutions as `GA`. However, `GA+` is significantly faster than `GA` in all assignment instances. On average, `GA+` is 19-, 16-, 23-, 22-, and 38-times faster than `GA` in 4-, 8-, 16-, 24-, and 32-way assignments, respectively. For the 16-way assignment of the largest dataset `ClueWeb-A`, `GA` finds a solution in about 23 days while `GA+` finds the same solution in less than four hours, i.e., `GA+` runs about 154 times faster than `GA` for that assignment instance.

4.6.2.2 General Comparison

For general performance comparison, we will only consider `MinMin+`, `MaxMin+`, `Suff+`, `GA+`, and `RC` since the improved versions perform better than their traditional counterparts and `MaxMin+` performs significantly better than `RASA`.

For the six skewed datasets, both of the proposed hybrid algorithms, `MaxMin+` and `Suff+`, find considerably better solutions than `MinMin+`, in terms of load balancing quality. Out of 30 assignment instances of skewed datasets, `RC`, `MaxMin+`, and `Suff+` find the best solutions in 14, 11, and 11 assignment instances, respectively. As seen in Table 4.8, `MaxMin+` and `Suff+` respectively attain load

imbalance values of 177.74% and 178.31% compared to 177.26% of RC, on average. Hence, MaxMin+ and Suff+ display comparable performance with RC in terms of load balancing quality. However, both MaxMin+ and Suff+ are significantly faster than RC in all of these 30 assignment instances. On average, MaxMin+ and Suff+ respectively run 2657- and 1588-times faster than RC. Hence, the use of RC in large datasets is not feasible.

For skewed datasets, we recommend the use of MaxMin+. Because, as seen in Tables 4.8 and 4.14, MaxMin+ is considerably faster than Suff+ and yields comparable performance in terms of load balancing quality.

For the 13 non-skewed datasets, GA+ finds the best solutions in 51 assignment instances out of 65 assignment instances in terms of load balancing quality. GA+ performs better than the other heuristics in assignment instances where MinMin+ already shows good performance (e.g., SpMxV and DVR datasets). This can be attributed to the fact that GA+ improves the initial assignment provided by MinMin+. Furthermore, GA+ is approximately two orders of magnitude slower than MinMin+. Hence, to analyze the performance of MinMin+, we exclude GA+ in the statistics given in the following paragraph to show the relative performance of the algorithms in finding the best assignments.

Out of 65 assignment instances of the non-skewed datasets, RC, MinMin+, MaxMin+, and Suff+ find the best assignments in 17, 17, 18, and 17 assignment instances, respectively. As seen in Table 4.8, MinMin+, MaxMin+ and Suff+ respectively attain load imbalance values of 0.62%, 0.62%, and 0.51% compared to 0.61% of RC, on average. Hence, MinMin+, MaxMin+, and Suff+ display comparable load-balancing performance with RC for non-skewed datasets. However, for these 65 assignment instances, MinMin+, MaxMin+, and Suff+ respectively run 2229-, 499-, and 236-times faster than RC, on average. Hence, the use of RC is not feasible also for large non-skewed datasets. For these 65 assignment instances, MinMin+ runs 13- and 52-times faster than MaxMin+ and Suff+, respectively, on average. We observe a trade-off between the solution quality and running times of MinMin+ and GA+. GA+ displays better load balancing performance than MinMin+, whereas MinMin+ is significantly faster (110-times, on average).

For non-skewed datasets, we recommend the use of `MinMin+`, since `MinMin+` runs significantly faster than both `MaxMin+` and `Suff+` while achieving comparable load balancing performance. The use of `GA+` should be considered only if the significantly higher running time of `GA+` can be amortized by the improved load balancing on the target application.

Chapter 5

Geographically Distributed Web Crawling: A Task Assignment Approach

5.1 Web Crawling and Independent Task Assignment

Web crawling is the process of locating, fetching, and storing the content available in the Web [87]. It finds application in a wide range of areas including web search engines, web archival systems, and data acquisition tasks involving web data mining and processing. Depending on the application area, the performance objectives of a web crawling system vary. For example, for a commercial web search engine, primary objectives could be to cover a large fraction of the content available in the Web and to keep the obtained content as fresh as possible [15]. For a data acquisition application (e.g., a focused web crawler that downloads topic-specific content [22]), however, the speed at which the crawler discovers relevant content may be a relatively more important objective.

A performance objective that is common to most web crawling applications

is to attain high download speed, i.e., to maximize the amount of content fetched from the Web per unit of time. Perhaps, the best example for this objective is an archival system that continuously takes snapshots of web pages in time with the goal of maximizing the number of distinct snapshots in its repository. In certain cases, the download speed of a web crawling system may also have an impact on the quality objectives [16, 44]. For example, as its download speed increases, the crawling system can download more web pages, increasing its content coverage, or can refetch downloaded pages more often, increasing the freshness of its content repository.

So far, many attempts have been made to improve the efficiency of web crawlers. Based on the type of parallelism employed within the crawling architecture, these efforts can be broadly classified under four headings, listed here in increasing order of complexity: single-threaded, multi-threaded, multi-node, and multi-site web crawling architectures. In its simplest form, a crawler is a single-thread process that discovers and fetches the content by following the hyperlink information within pages. Obviously, due to the sequential nature of content acquisition, this type of crawling does not scale well in terms of the download throughput. The possibility of fetching pages, concurrently, from different web servers leads to multi-threaded crawling [58]. In this approach, the network bandwidth available to the crawler is tried to be saturated until the context switching overhead due to multi-threading becomes the main bottleneck. A slightly more complex approach is to parallelize the crawling process over a multi-node system (e.g., a cluster of computers) [25]. This approach allows scalability in terms of multi-threading and memory consumption, with little parallelization overhead in the form of inter-processor communication. Indeed, today's large-scale web crawlers (e.g., those used in commercial web search engines) are massively parallel systems located in a large data center. A natural extension to parallelizing the crawling process within a single data center is to distribute the process over multiple data centers that are geographically distant to each other [17]. Compared to the centralized crawling architectures, geographically distributed web crawling offers improved fault tolerance due to its resilience to network partitions, better coupling with geographically distributed indexing, and higher download speeds

since the network proximity can be exploited.

In this chapter, we concentrate on geographically distributed web crawling architectures. In theory, such architectures may be highly distributed and involve thousands or millions of crawlers (e.g., P2P web crawling). Herein, however, we restrict our scope to a case where crawling is performed by several large-scale data centers that are geographically distant to each other. We find this scenario more interesting due to potential scalability issues in P2P-like web crawling and the recent research interest in multi-site web search engines [6, 16, 19].

In our setting, we assume that web servers are uniquely assigned to crawlers to prevent redundant download of the same page by multiple crawlers. Hence, each crawler independently crawls the content hosted on servers assigned to itself. We also assume that, due to the potential variation in the hardware resources available in data centers, web crawlers can be heterogeneous in terms of their bandwidth and processing capacities. Finally, we take into account the fact that a crawler's download speed negatively correlates with its geographical proximity to web servers, i.e., web crawlers should crawl the content hosted by nearby web servers [17].

The above-mentioned assumptions and requirements lead to the geographically distributed web crawling problem, where the goal is to assign servers to crawlers so that the download speed of the crawling system is maximized. We show that this problem can be formulated as a task assignment problem, which forms the focus of this chapter. In particular, we make the following contributions. We introduce two variants of the task assignment problem for geographically distributed web crawling architectures. We adapt several task assignment algorithms taken from the literature to one of these problems. Finally, we conduct experiments using real-life web data collections and network statistics. The obtained results demonstrate the potential performance improvements that can be attained by the proposed task assignment approach over a relatively naive baseline.

The rest of the chapter is organized as follows. In Section 5.2, we provide an

Table 5.1: The notation used in this chapter

Symbol	Description
A	server-to-crawler assignment vector
$A[i]$	assignment decision of server S_i to crawler $C_{A[i]}$
B_k	bandwidth of crawler C_k
$\mathcal{C}$	set of web crawlers
C_k	kth web crawler
G	number of chromosomes in the genetic algorithm
H	number of iterations in the genetic algorithm
K	number of crawlers
L_k	geographical location of crawler C_k
M	makespan
N	number of web servers
$\mathcal{P}$	set of processors
P_k	kth processor
$R_{i,k}$	round-trip network latency between S_i and C_k
$\mathcal{S}$	set of web servers
S_i	ith web server
$\mathcal{T}$	set of tasks
T_i	ith task
$t_{i,k}$	estimated time for C_k to crawl the content in S_i
U	a set of servers
X	expected-time-to-crawl matrix
b	index of the bottleneck crawler
c	speed of light on copper wire
$d_{i,k}$	geographical distance between S_i and C_k
e_k	current load of C_k
i, j	indices that refer to servers
k, ℓ	indices that refer to crawlers
n_i	number of pages on server S_i
w_i	amount of content in bytes on server S_i
$x_{i,k}$	time required for crawler C_k to crawl server S_i

overview of the multi-site web crawling architecture that we consider in this chapter and formally state the investigated task assignment problem for geographically distributed web crawling. Several task assignment heuristics are discussed as potential solutions in Section 5.3. In Section 5.4, we describe the cost model used in simulations. Section 5.5 presents experimental findings regarding the crawling performance. In Section 5.6, we provide a brief survey of the related work on distributed web crawling. Table 5.1 summarizes the notation that we will use in the rest of the chapter.

5.2 Server-to-Crawler Assignment Problem

5.2.1 Architecture

The envisioned crawling architecture involves a number of fixed-location data centers that are geographically distant to each other (e.g., one data center per major continent). Each data center is equipped with a limited amount of hardware resources, i.e., bandwidth, storage, and computing power. Data centers can be heterogeneous in terms of the amount and type of their hardware resources. Each data center hosts a separate web crawler, which potentially runs on a large cluster of computers.

In our architecture, a crawler fetches only the pages hosted by the web servers assigned to itself. This leads to a natural partitioning of web servers as illustrated in Fig. 5.1. The crawlers coordinate to avoid duplicate crawling of the same pages, simply by exchanging the URLs, i.e., whenever a crawler discovers a URL belonging to a web server that is assigned to a remote crawler, the URL is communicated to that remote crawler where it will be fetched. We assume that the main performance bottleneck in crawling is the network, i.e., sufficient processing power and storage is always available. We also assume that the overhead of link exchanges is negligible. These assumptions will shape the cost model that we will describe in Section 5.4.

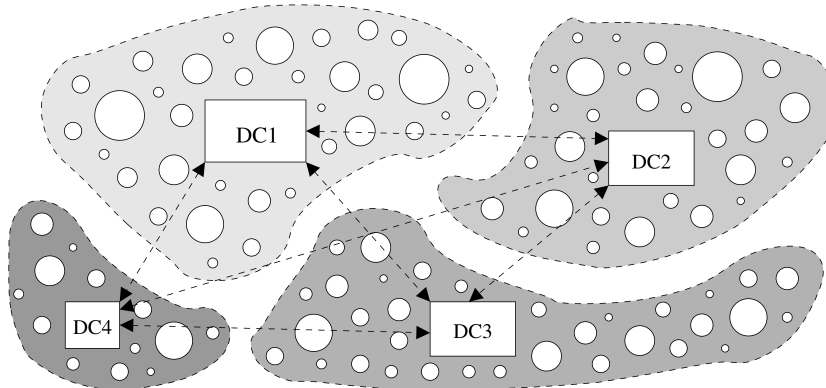


Figure 5.1: A geographically distributed web crawling architecture with four data centers (DC1–DC4), each crawling a non-overlapping subset of web servers (the circles in the figure) in the Web.

In practice, depending on the application, crawled pages may be locally processed without any coordination between the data centers (e.g., a data mining application) or may be redistributed/replicated for indexing based on the past usage patterns (e.g., a web search engine with a geographically distributed index [7, 11]). To be as general as possible, we do not tie our architecture to a particular application. Hence, herein, we are not interested in the efficiency of potential processing tasks which may follow the crawling process.

Finally, we note that our focus is not on incremental web crawling, where the crawling system continuously crawls the web and tries to extend its frontier by discovering new pages. We rather focus on a specific case where a fixed number of servers are tried to be crawled (or re-crawled) by a geographically distributed crawler. The closest use cases are a web archival system that periodically takes snapshots of the Web or a search engine that aims to maintain the freshness of its repository by re-crawling the same web content.

5.2.2 Notation

We are given a set of web servers $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$ and a set of web crawlers $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$.[1] Each server $S_i \in \mathcal{S}$ is associated with two weights w_i and n_i, which represent the amount of content (in bytes) and the number of pages hosted by the server, respectively. Each crawler $C_k \in \mathcal{C}$ is associated with two attributes: the geographical location L_k of the data center where the crawler is running and the network bandwidth B_k available to the crawler.

For every server and crawler pair (S_i, C_k), we are given the time cost incurred to C_k if all content hosted in S_i is downloaded by C_k, i.e., we have an expected-time-to-compute (ETC) matrix

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,K} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,K} \end{pmatrix}, \tag{5.1}$$

where $x_{i,k}$ denotes the estimated time for crawler C_k to crawl the content of server S_i. Note that, for the web crawling problem, the "compute" term means "crawl". The expected-time-to-compute matrix can also be read as expected-time-to-crawl matrix.

In parallel computing literature, the ETC matrices are classified into two categories: consistent and inconsistent [1,69]. In our context, the consistent ETC matrices imply an ordering on the speed of crawlers. If a crawler C_k is faster than another crawler C_ℓ in crawling a given web server, then C_k is faster than C_ℓ in crawling any other web server as well. The inconsistent ETC matrices do not imply any ordering. A crawler may be the fastest to crawl a given web server, yet there may be a faster crawler for another server.

In our problem, the constructed ETC matrix is potentially inconsistent. For two crawlers C_k and C_ℓ, the ordering between $x_{i,k}$ and $x_{i,\ell}$ for a server S_i does not

[1] Throughout the text, the i and j indices will be used to refer to the servers while the k and ℓ indices will be used to refer to the crawlers.

imply any ordering between $x_{j,k}$ and $x_{j,\ell}$ for another server S_j. In other words, the crawler that can crawl a server in the shortest amount of time varies from server to server. In the rest of the chapter, we refer to the fastest crawler for a server as the favorite crawler of that server.

5.2.3 Problem

In this section, we present two variants of the server-to-crawler assignment problem for geographically distributed web crawlers.

5.2.3.1 Minimizing Total Crawling Time

In this version of the problem, the optimization objective is to assign the web servers to the crawlers such that the total time spent by the system to download the content of web servers is minimized. This problem can be formally defined as an integer linear programming model:

$$\text{minimize} \sum_{i,k} \{x_{i,k} \times \beta_{i,k}\}$$

such that

$$\beta_{i,k} \in \{0,1\}, \quad \text{for each } S_i \in \mathcal{S}, C_k \in \mathcal{C};$$
$$\sum_{k} \beta_{i,k} = 1, \quad \text{for each } S_i \in \mathcal{S},$$

where the decision variable $\beta_{i,k}$ indicates whether S_i is assigned to C_k, i.e., $\beta_{i,k}$ is 1 if and only if S_i is assigned to C_k, or 0, otherwise.

This problem instance can be solved by means of a greedy algorithm: Minimum Execution Time (`MET`) [5,10,48], which assigns each server S_i to its favorite crawler. In other words, for each server S_i, $\beta_{i,k} = 1$ such that $k = \text{argmin}_k\, x_{i,k}$. `MET` is an exact algorithm, which finds an optimum solution, for this variant of the problem. An $O(KN)$-time algorithm for `MET` is presented in Algorithm 5.1, where $A[i]$ denotes the index of the crawler responsible for crawling server S_i.

Algorithm 5.1 MET(X, K, N)

```
1: for i ← 1 to N do
2:     min ← ∞
3:     for k ← 1 to K do
4:         if x_{i,k} < min then
5:             min ← x_{i,k}
6:             k' ← k
7:     A[i] ← k'
8: return A
```

5.2.3.2 Minimizing Maximum Crawling Time

In this version of the problem, the objective is to minimize the workload of the maximally loaded crawler (bottleneck crawler). This objective effectively minimizes the duration of the crawling. As a solution, a formal integer linear programming formulation is presented below:

$$\text{minimize} \quad \max_k \sum_i x_{i,k} \times \beta_{i,k}$$

such that

$$\beta_{i,k} \in \{0,1\}, \quad \text{for each } S_i \in \mathcal{S}, C_k \in \mathcal{C};$$
$$\sum_k \beta_{i,k} = 1, \quad \text{for each } S_i \in \mathcal{S}.$$

This optimization problem can be formulated as an independent task assignment problem in heterogeneous computing systems. In this problem, we have a set $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$ of N independent tasks, a set $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ of K heterogeneous processors, and an expected-time-to-compute matrix $X = \{x_{i,k}\}_{N \times K}$, where $x_{i,k}$ denotes the execution cost of task T_i on processor P_k. The objective is to find a task-to-processor assignment that results in the minimum turnaround time (makespan). In our problem context, T_i corresponds to the task of crawling server S_i, P_k corresponds to the crawler C_k, and $x_{i,k}$ corresponds to the time required for crawler C_k to crawl server S_i.

As mentioned earlier, the `MET` algorithm provides an exact solution to the first

Algorithm 5.2 MCT(X, K, N)

```
1: for k ← 1 to K do
2:     e_k ← 0
3: for each i ∈ {1, ..., N} in random order do
4:     min_makespan ← ∞
5:     for k ← 1 to K do
6:         if e_k + x_{i,k} < min_makespan then
7:             min_makespan ← e_k + x_{i,k}
8:             k′ ← k
9:     A[i] ← k′
10:    e_{k′} ← e_{k′} + x_{i,k′}
11: return A
```

problem variant. We believe that from a research perspective, the second problem variant is more complex and interesting. Moreover, it is more important in terms of its implications in practice. Hence, in the rest of the chapter, we focus only on the problem of minimizing the maximum crawling time.

5.3 Heuristic Solutions

In this section, we provide the most notable heuristics that are proposed in the literature as a solution for the independent task assignment problem. This problem has been previously studied in the domain of parallel and distributed computing [10, 35, 39, 63, 77, 97, 103, 107]. The problem is known to be NP-complete [63].

For the sake of a clear presentation, we describe the heuristics using the notation we introduced before. For each heuristic, we provide an algorithm which computes a server-to-crawler assignment in the form of a vector A of size N. A vector element $A[i] = k$ denotes the assignment of server S_i to crawler C_k. In all algorithms, the e_k variable keeps track of the load that is currently assigned to crawler C_k. All heuristics mentioned in this section (except for the genetic algorithm) are constructive in nature, and they are based on greedy choices that depend on the momentary completion times of crawling tasks.

Algorithm 5.3 PPB(X, K, N)

```
1:  y ← ⌈p × K/100⌉
2:  for k ← 1 to K do
3:      e_k ← 0
4:  for each i ∈ {1, ..., N} in random order do
5:      min_makespan ← ∞
6:      τ ← SELECT(x_i, y)
7:      for k ← 1 to K do
8:          if x_{i,k} ≤ τ and e_k + x_{i,k} < min_makespan then
9:              min_makespan ← e_k + x_{i,k}
10:             k' ← k
11:     A[i] ← k'
12:     e_{k'} ← e_{k'} + x_{i,k'}
13: return A
```

Minimum Completion Time (MCT) [5, 10]: This heuristic iterates over the servers in an arbitrary order and assigns each server to a crawler that will result in the minimum completion time for that server. An $O(KN)$-time algorithm for MCT is presented in Algorithm 5.2.

p-percent Best (PPB) [78]: The heuristic is similar to the MCT algorithm. PPB limits the assignment of servers only to the most favorite crawlers in the top p-percent of crawlers, where $0 < p \leq 100$. When $p = 100/K$, PPB is equivalent to MET. When $p = 100$, PPB is equivalent to MCT. An $O(KN)$-time algorithm for PPB is presented in Algorithm 5.3. In the algorithm, SELECT refers to a function that returns the yth smallest element in an array of values [34].

MinMin+ (MinMin+): The MinMin heuristic has been widely used in the literature as a benchmark to judge the performance of more recently proposed heuristics. This heuristic performs N iterations, at each iteration, selecting an unassigned server and assigning it to a crawler. The assignment decision is made based on a two-step procedure. In the first step, a unique crawler is identified for each unassigned server. The crawler is selected such that it will finish the download of the server's content at earliest time point (the current workloads of crawlers are also taken into account), i.e., achieve the minimum completion

time (MCT). In the second step, the server with the minimum MCT is selected and assigned to the crawler identified in the first step. In the `MinMin` heuristic, the assignment of servers with little content are performed in earlier iterations while servers hosting large amounts of content are assigned later. We reduced the time complexity of `MinMin` from $O(KN^2)$ to $O(KN \log N)$ without altering the solution quality of the resulting assignments at all and achieving significant runtime improvements. Details of `MinMin+` can be found in Chapter 4.

MaxMin+ (`MaxMin+`): In the `MaxMin` heuristic, the servers are assigned to crawlers after a two-step procedure, somewhat similar to that in `MinMin`. The main difference is in the server selection policy in the second step, where `MaxMin` selects the server with the maximum MCT (instead of the minimum MCT) and assigns it to the crawler identified in the first step. Due to this selection policy, `MaxMin` favors the assignment of servers with large amounts of content in earlier iterations. `MaxMin` is an $O(KN^2)$-time heuristic, which is too slow to be used in practice when the number of tasks is large. We have proposed `MaxMin+` heuristic, which is a hybrid between `MaxMin` and `MinMin+`, aiming to improve `MaxMin` in terms of both solution quality and running time. To this end, `MaxMin+` adopts the server selection policy of `MaxMin` or `MinMin`: When the load of the bottleneck crawler changes, `MaxMin+` behaves like `MaxMin`; otherwise, it behaves like `MinMin`. The details of `MaxMin+` heuristic is presented in Chapter 4.

Sufferage+ (`Suff+`): The `Suff` heuristic follows a similar procedure to that in `MaxMin` and `MinMin`. The main difference is in that, in the first step of the process, `Suff` computes the second MCT value for each server in addition to the MCT value. In the second step, a sufferage value, which is defined as the difference between the MCT and the second MCT values, is calculated for each server. The `Suff` heuristic then selects the server with the largest sufferage value and assigns it to the crawler identified in the first step. Like `MaxMin`, `Suff` is an $O(KN^2)$-time heuristic, and this quadratic running time complexity prevents its application to problems with large number of tasks. The recently proposed `Suff+` heuristic is a hybrid between `Suff` and `MinMin+`. `Suff+` adopts the selection criterion of `Suff` whenever there will be an increase in the makespan or, otherwise, the selection criterion of `MinMin+`. This way, `Suff+` combines the speed of `MinMin+` and the

quality of `Suff` to obtain a faster heuristic that can be applied to large datasets. The details of `Suff+` can be found in Chapter 4.

Genetic Algorithm (`GA`): The use of `GA` for the solution of the independent task assignment problem is described in [10]. In our problem setting, each chromosome represents a different server-to-crawler assignment. Assuming G chromosomes, one of the chromosomes is initially populated with `MinMin` while the remaining $G-1$ chromosomes are populated with random assignments. Maintaining the best assignment guarantees that the solution quality of `GA` is not worse than the quality of `MinMin`. Crossover is implemented as a single random cross on the paired chromosomes. Mutation is defined as reassigning a random server to a random crawler. A faster `GA` algorithm, referred to as `GA+`, is obtained by replacing the initial solver `MinMin` with faster `MinMin+`. The result is an asymptotically faster algorithm. Details of `GA+` algorithm is presented in Chapter 4.

5.4 Cost Model for Crawling Times

The task assignment heuristics described in Section 5.3 assume the availability of an expected-time-to-crawl matrix. That is, the time cost $x_{i,k}$ for crawler C_k to download the content on server S_i is assumed to be known, for every (crawler, server) pair. In practice, the $x_{i,k}$ values can be approximated by sampling pages from web servers, by exploiting the information obtained in the previous crawling sessions, or by relying on external services, such as Alexa[2]. In our chapter, since we are conducting simulations, we rely on an analytical cost model. Although the accurate estimation of the $x_{i,k}$ values is not the main focus of this chapter, we try to devise a realistic cost model, as much as possible.

In our cost model, the time required for crawler C_k to download the content of server S_i is estimated as

$$x_{i,k} = R_{i,k} + t_{i,k}. \tag{5.2}$$

Here, $R_{i,k}$ represents the total round-trip network latency for establishing the

[2] `http://www.alexa.com`.

TCP and HTTP connections between crawler C_k and server S_i while $t_{i,k}$ denotes the time that C_k spent while retrieving the content of S_i over the network. Essentially, the first summation term accounts for the propagation delay while the second term accounts for the transfer time. If we assume that queueing delays in the routers is negligible, these are the two common overheads in retrieving content over the network.

When estimating the $R_{i,k}$ values, we assume that the number of connections that a crawler establishes to a server is proportional with the number of pages in the server. This is because, in practice, the crawlers are expected to close the established connection each time after a page is retrieved, due to the politeness constraints. Hence, we write $R_{i,k}$ as

$$R_{i,k} = 2 \times n_i \times r_{i,k}, \tag{5.3}$$

where n_i denotes the number of pages in server S_i and $r_{i,k}$ is the network latency between C_k and S_i. To estimate the $r_{i,k}$ values, we rely on the great-circle distance estimation technique, assuming that the network latency correlates well with geographical distance [61]. We estimate $r_{i,k}$ as

$$r_{i,k} = c_1 + c_2 \times \frac{d_{i,k}}{c}, \tag{5.4}$$

where c denotes the speed of light on copper wire (we use 200,000 km/s) and $d_{i,k}$ denotes the great-circle distance between C_k and S_i. Here, c_1 and c_2 are constants estimated by linear regression over sample latency values observed in real life. Following [19], we use $c_1 = 8.239$ ms and $c_2 = 1.983$.

To calculate the $d_{i,k}$ values, we use an IP-to-Geo database [80], which claims to have an accuracy of 99.5% at the country level. For every server in our data, using its IP address, we identify the country where the web server is hosted. The great-circle distance between a server and a crawler is then approximated by the distance between the capitals of the home countries hosting the server and the crawler. We also add a constant value (set to 637 km) to account for the distance between the capital and the actual city where the server is hosted.

The second term in Eq. (5.2) depends on the amount of content to be downloaded from the server and the network bandwidth between the crawler and the server. Herein, we make the simplifying assumption that the network bandwidth between the crawler and any web server is determined by the download bandwidth of the crawler. Hence, the time for crawler C_k to download the content of server S_i is estimated as

$$t_{i,k} = \frac{w_i}{B_k}, \tag{5.5}$$

where w_i is the amount of content (in bytes) on server S_i and B_k is the bandwidth of the crawler C_k.

In our cost model, we also take into account the freshness requirement of each server, i.e., how frequently the pages of a server need to be crawled. In practice, this is an application-specific parameter. For example, an archiving crawler may give equal opportunity to each server, whereas a commercial search engine crawler may prefer to recrawl pages on highly visited web servers more often.

In our model, we associate each server S_i with a parameter f_i that indicates the frequency at which pages of S_i are crawled. As an illustrative case, we assume that the top-level pages in the directory hierarchy should be crawled more often and compute the f_i value as

$$f_i = \frac{n_i}{n_i + \sum_{p=1}^{n_i} s_{i,p}}, \tag{5.6}$$

where $s_{i,p}$ denotes the depth of a particular page p on server S_i in the directory hierarchy (i.e., the number of path separator characters in the URL of the page).

Given Eqs. (5.2)–(5.5), the total time that crawler C_k needs to crawl server S_i can now be written as

$$x_{i,k} = f_i \times 2 \times n_i \times \left(c_1 + c_2 \times \frac{d_{i,k}}{c}\right) + f_i \times \frac{w_i}{B_k}. \tag{5.7}$$

We note that the second term in Eq. (5.7) decreases as the network bandwidth of the crawlers increases. However, the first term is limited with the speed of light and the characteristics of the physical network. Hence, the distance between

Table 5.2: Properties of the datasets

Dataset	# of servers	# of pages	# of countries	Content size
`ClueWeb-B`	0.8 million	49.2 million	215	1.4 TB
`ClueWeb-A`	2.5 million	2.5 billion	232	28.3 TB
`YWC`	137.7 million	5.4 billion	248	229.3 TB

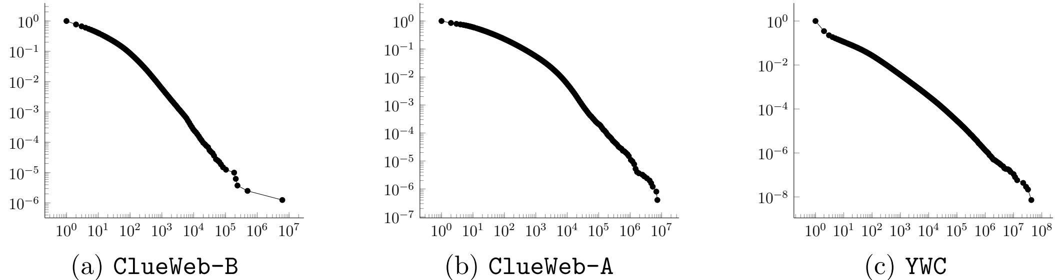


Figure 5.2: The cumulative density distribution for the number of pages on servers. x-axis (log scale): number of pages on servers, y-axis (log scale): cumulative density distribution, i.e., $P(X \geq x)$.

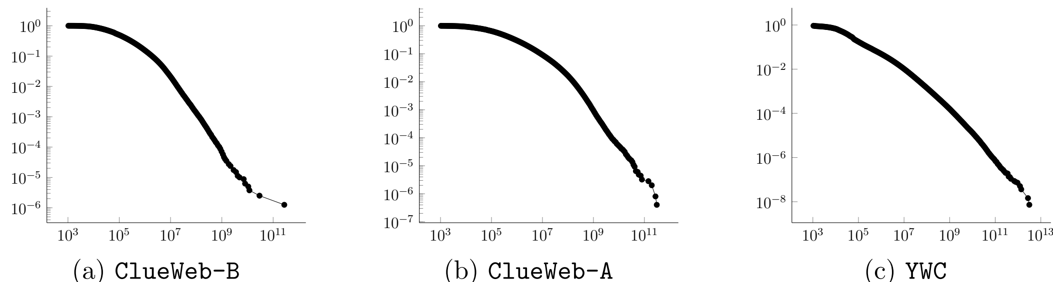
the crawlers and web servers becomes more important with increasing network bandwidth.

5.5 Experiments

5.5.1 Datasets

We conduct our simulations on three different web datasets: `ClueWeb-B`, `ClueWeb-A`, and `YWC` (listed in increasing order of their size). The first two datasets are part of the ClueWeb09 web page collection [31], which is obtained through a large web crawl performed in 2009. `ClueWeb-A` contains all pages in the original collection while `ClueWeb-B` contains a smaller subset of English pages. `YWC` is the result of a considerably larger web crawl performed by Yahoo! in 2011. The properties of the three datasets are displayed in Table 5.2.

Fig. 5.2 shows the cumulative density distribution for the number of web pages on servers. In all datasets, we observe a highly skewed page distribution,



(a) `ClueWeb-B` (b) `ClueWeb-A` (c) `YWC`

Figure 5.3: The cumulative density distribution for the amount of content (in bytes) on servers. x-axis (log scale): amount of content on servers, y-axis (log scale): cumulative density distribution, i.e., $P(X \geq x)$.

i.e., there are many servers with few pages, but few servers with many pages. Similarly, Fig. 5.3 provides the cumulative density distribution for the content sizes of servers. We observe this distribution to be even more skewed. These plots indicate high variation in task sizes, implying that our task assignment problem is a relatively difficult one.

In Table 5.3, we display the distribution of web content on countries with the largest presence in each dataset. The second column of each table, for a different dataset, indicates the amount of content hosted by the web servers located in a particular country. A bias in the content distribution can be noticed for the `ClueWeb-B` and `YWC` datasets. The web pages in `ClueWeb-B` are skewed towards European countries, potentially due to the small scale of this dataset and the fact that it is limited to English pages (e.g., in the Netherlands, there are large web hosts serving Wikipedia pages[3]. In case of `YWC`, we observe a strong bias towards Asian countries. `ClueWeb-A` does not seem to have such a strong bias.

5.5.2 Centralized Crawling Performance

Table 5.4 shows the crawling performance for a centralized crawler located in a particular country (the countries are limited to those in Table 5.3). In the table, the "Days" column shows the number of days needed by a crawler located in a country to download all of the pages in a dataset (e.g., it takes 47 days for a

[3] `http://www.wikipedia.org`

Table 5.3: Distribution of web content on countries

Country	Size (GB)
US	840.7
Netherlands	271.5
UK	52.2
Germany	40.4
Bulgaria	31.2
Canada	29.2
France	27.9
Italy	13.0
Australia	9.1
Turkey	7.8
China	7.4
Japan	6.8
Spain	6.5
Korea	5.5
Switzerland	4.7
Ireland	3.3

(a) `ClueWeb-B`

Country	Size (GB)
US	11,930.8
France	5,852.8
China	3,087.7
Germany	1,731.8
Japan	1,195.6
UK	784.0
Spain	717.7
Netherlands	575.5
Korea	529.7
Italy	521.0
Canada	466.3
Bulgaria	350.6
Brazil	308.2
Switzerland	131.8
Australia	119.3
Taiwan	108.2

(b) `ClueWeb-A`

Country	Size (GB)
US	76,237.4
Japan	31,445.3
Korea	15,584.4
China	12,695.8
Taiwan	11,830.8
UK	10,378.7
France	7,211.7
Germany	6,801.6
Spain	6,491.9
Italy	4,892.2
Hong Kong	3,866.2
Brazil	3,165.7
Russia	3,050.9
Canada	2,802.1
Australia	2,370.2
Vietnam	1,637.4

(c) `YWC`

crawler located in the US to download all pages in the `ClueWeb-B` dataset). The last column of the tables shows the relative performance of a crawler with respect to the best crawler for a particular dataset. According to the table, the crawler located in the US is the fastest crawler for all datasets. This is mainly because this country hosts large amounts of web content and its geographical location is suitable for faster web crawling.

Fig. 5.4 demonstrates the impact of assigning a server to a faster crawler. The bottom curve in the figures represents the cumulative density distribution for the total download time, assuming that a server is always assigned to its favorite crawler (the one that can crawl a server in the shortest amount of time). The top curve shows the same information, but assuming that each server is assigned to the crawler with the worst performance for that server. The curve in the middle represents the average performance over all possible crawlers. On average, we observe an order of magnitude difference between the crawling times of the best and worst crawlers of a server. We also observe that the average crawling time is about only half of that of the worst crawler. This is expected

Table 5.4: Performance of a centralized crawler located in a particular country

Country	Days	%Δ
US	47	0.0
Canada	61	30.8
Turkey	63	35.8
UK	67	44.1
Germany	69	47.5
Netherlands	83	78.1
China	84	80.9
Spain	101	117.9
Australia	102	119.2
Switzerland	107	128.7
Korea	157	237.1
Japan	182	291.1
Italy	187	301.7
Bulgaria	189	305.2
France	204	338.9

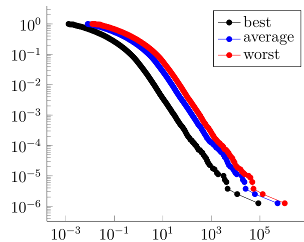
(a) `ClueWeb-B`

Country	Days	%Δ
Canada	2,142	0.0
UK	2,647	23.6
US	2,717	26.8
China	3,514	64.0
Germany	3,890	81.6
Netherlands	3,964	85.1
Australia	4,806	124.3
Spain	4,832	125.5
Brazil	5,883	174.6
Switzerland	5,883	174.6
Korea	7,509	250.5
Japan	9,668	351.3
Bulgaria	10,153	374.0
Italy	10,173	374.9
France	11,029	414.8

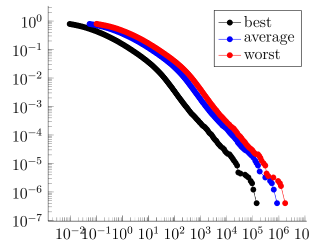
(b) `ClueWeb-A`

Country	Days	%Δ
US	10,719	0.0
Germany	11,635	8.5
Canada	11,750	9.6
UK	11,760	9.7
China	12,215	13.9
Spain	12,307	14.8
Brazil	12,832	19.7
Hong Kong	13,234	23.5
Australia	13,751	28.3
Japan	13,931	30.0
Russia	14,552	35.8
Taiwan	14,572	35.9
Korea	14,678	36.9
Italy	15,458	44.2
France	15,853	47.9

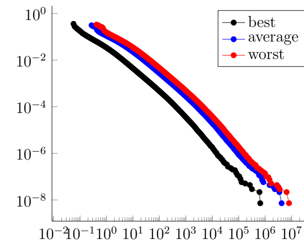
(c) `YWC`



(a) `ClueWeb-B`



(b) `ClueWeb-A`



(c) `YWC`

Figure 5.4: Log-log plots of the cumulative density distribution of best, average, and worst crawler download times of servers in seconds. x-axis: download times, y-axis: cumulative density distribution, i.e., $P(X \geq x)$.

because the crawlers are somewhat evenly distributed over the globe. In general, we find that the distance between a web server and its best performing crawler is relatively small when compared to the average distance between the server and other crawlers.

5.5.3 Performance of Task Assignment Heuristics

Crawler placement. Our simulations require assigning a fixed number of crawlers to different geographical locations.[4] To this end, we adopt a simple approach and virtually place the crawlers in the top K countries with highest presence in a dataset. For example, in a simulation using four crawlers and the `ClueWeb-A` dataset, the crawlers are assumed to be located in the United States, France, China, and Germany, i.e. the top four countries in Table 5.3(b).

The simulations are conducted with four different crawler counts: $K = 2, 4, 8, 16$, for each dataset. Each dataset and K combination forms a different assignment instance in our simulations. Since we have three datasets and four different K values, we have a total of 12 assignment instances.

Baseline. We compare the heuristics presented in Section 5.3 against a simple baseline. We simply generate virtual Voronoi cells for each crawler on the surface of the Earth and assign each server to the geographically closest crawler. If the bandwidth of the crawlers are assumed to be equal, the cost model we described in Section 5.4 implies that the crawler that obtains the minimum $x_{i,k}$ value for a server is the one with the smallest $d_{i,k}$ value. In that case, this baseline becomes equivalent to the `MET` heuristic, described in Section 5.2. This is illustrated below.

$$\begin{aligned}
\operatorname*{argmin}_k x_{i,k} &= \operatorname*{argmin}_k \left\{ 2 \times n_i \times \left(c_1 + c_2 \times \frac{d_{i,k}}{c} \right) + \frac{w_i}{B_k} \right\} && (5.8)\\
&= \operatorname*{argmin}_k \left\{ 2 \times n_i \times \left(c_1 + c_2 \times \frac{d_{i,k}}{c} \right) \right\} && (5.9)\\
&= \operatorname*{argmin}_k \{ d_{i,k} \} && (5.10)
\end{aligned}$$

[4]In this work, we are not interested in the dual problem of finding the best possible crawler locations.

Setup. The heuristics are implemented in the Java programming language. The simulations are carried out on a Linux workstation equipped with six 2100-MHz quad-core CPUs and 132 GB of memory. Since the `MCT`, `PPB` and `GA+` heuristics involve non-deterministic components, these heuristics were executed 10 times with different random seeds for each assignment instance and the average performance figures are reported. Due to the high running time and extremely high memory requirements of the `GA+` heuristic, it was not feasible to complete the simulations on the `YWC` dataset for this heuristic. The incomplete results are indicated by "–" in the tables.

Load balancing performance. One of the important performance metrics is the imbalance in the total crawling times of the crawlers. We define the load imbalance %LI of an assignment heuristic relative to the ideal makespan as

$$\%\text{LI} = \frac{M - M^*}{M^*}, \tag{5.11}$$

where M denotes the makespan of the assignment produced by the heuristic and M^* denotes the ideal makespan for the given assignment instance. The ideal makespan is computed as

$$M^* = \frac{W^*_{\text{tot}}}{K} = \frac{\sum_i \min_k\{x_{i,k}\}}{K}, \tag{5.12}$$

where W^*_{tot} is a lower bound on the total crawling time that would be attained if each server was assigned to its favorite crawler. Note that the M^* value is a rather loose lower bound on the makespan, i.e., the optimal makespan is very likely to be greater than M^*.

Table 5.5 displays the load imbalance values for the 2-, 4-, 8-, and 16-way assignments produced by the evaluated task assignment heuristics. The bold values in each row indicate the best performing heuristic(s) for the corresponding assignment instance. As seen in Table 5.5, the baseline leads to relatively large imbalance values, especially when K is high. The `PPB` and `MCT` heuristics display similar performance. There are, however, some performance differences in certain assignment instances. For example, the load imbalance of `PPB` is much lower than that of `MCT`, for the ($K = 4$, `ClueWeb-A`) assignment instance. The attained load

Table 5.5: Percent load imbalance values

Test	K	Baseline	Task assignment heuristics					
			PPB	MCT	MinMin+	MaxMin+	Suff+	GA+
ClueWeb-B	2	36.42	36.42	26.20	16.93	**13.46**	14.38	14.42
	4	48.69	48.45	48.34	63.90	**18.16**	22.60	60.01
	8	330.56	164.43	166.60	208.47	**103.22**	104.62	199.73
	16	794.61	384.27	361.94	420.30	**244.72**	**244.72**	410.51
ClueWeb-A	2	49.64	49.64	44.26	16.80	21.05	21.02	**15.97**
	4	32.77	22.29	53.80	16.02	11.53	**11.31**	14.40
	8	216.18	79.36	86.02	76.81	**58.32**	61.19	74.32
	16	663.36	169.77	152.95	171.83	128.95	**127.51**	168.50
YWC	2	6.33	6.33	30.87	1.24	**0.47**	1.34	–
	4	23.09	27.01	53.01	7.58	**5.46**	5.80	–
	8	97.33	55.59	67.70	23.91	18.77	**17.76**	–
	16	367.85	110.73	110.71	96.74	**68.98**	69.54	–

imbalance values are 222.24%, 100.20%, and 96.19%, on average, for the baseline, MCT, and PPB heuristics, respectively.

According to the table, MinMin+ shows a relatively inferior performance. It has an average load imbalance of 93.38%, which is slightly better than that of PPB. We observe MaxMin+ to perform considerably better. It is the winning heuristic in eight out of 12 assignment instances. Suff+ is the closest heuristic to MaxMin+ in terms of load balancing performance. The average load imbalance values attained by MaxMin+ and Suff+ are 57.76% and 58.48%, respectively.

Since GA+ exploits the initial solutions provided by MinMin+, we may expect it to perform better than MinMin+. In our results, however, we observe only a small improvement. Excluding the YWC dataset, the average load imbalance for GA+ is 119.73% while it is 123.88% for MinMin+. GA+ can find the best solution for the ($K = 2$, ClueWeb-A) assignment instance. However, there are assignment instances where GA+ fails to find a comparable solution. For example, for the ($K = 16$, ClueWeb-B) assignment instance, GA+ finds a solution with a load imbalance of 410.51 while Suff+ and MaxMin+ have a load imbalance of 244.72, on average.

Makespan. As a better indicator of the crawling performance, in Table 5.6, we

Table 5.6: Expected crawling duration (in days)

Test	K	Baseline	Task assignment heuristics					
			PPB	MCT	MinMin+	MaxMin+	Suff+	GA+
ClueWeb-B	2	5.5	5.5	5.1	4.7	4.6	4.6	4.6
	4	2.7	2.7	2.7	3.0	2.2	2.2	2.9
	8	2.7	1.7	1.7	1.9	1.3	1.3	1.9
	16	2.6	1.4	1.3	1.5	1.0	1.0	1.5
ClueWeb-A	2	152.1	152.1	146.7	118.7	123.1	123.0	117.9
	4	51.2	47.1	59.3	44.7	43.0	42.9	44.1
	8	47.0	26.6	27.6	26.3	23.5	23.9	25.9
	16	46.8	16.5	15.5	16.7	14.0	13.9	16.5
YWC	2	1,158.8	1,158.8	1,426.3	1,103.4	1,095.0	1,104.4	–
	4	447.4	461.6	556.1	391.0	383.3	384.5	–
	8	278.0	219.2	236.2	174.5	167.3	165.9	–
	16	237.4	106.9	106.9	99.8	85.7	86.0	–

display the expected crawling times (in days) for the bottleneck crawlers in each assignment instance. The results in the table demonstrate how load imbalance affects the duration of crawling. As an example, for the ($K = 16$, YWC) assignment instance, the assignment generated by the baseline has an expected crawling time of 237.4 days while the assignment generated by MaxMin+ has an expected crawling time of only 85.7 days.

According to Table 5.6, increasing the number of crawlers considerably improves the performance. For example, for the ($K = 2$, YWC) assignment instance, MaxMin+ finds a solution with an expected crawling time of 1095.0 days while, for the ($K = 4$, YWC) assignment instance, the expected crawling time reduces to 383.3 days. This improvement is super linear since the expected crawling time has decreased by a factor of about 2.86 while K is doubled. In general, we observe similar super-linear improvements in large datasets and low K values. As K increases, the performance gains are lower since the assignment of very large servers begin to determine the makespan.

Execution time. Table 5.7 displays the running times of the heuristics. All heuristics are sufficiently fast to be useful in practice as their running times are much smaller compared to the gains in crawling times. In general, the running time of most heuristics increases with increasing K. The running times of MaxMin+

Table 5.7: Execution times (in seconds)

Test	K	Baseline	Task assignment heuristics					
			PPB	MCT	MinMin+	MaxMin+	Suff+	GA+
ClueWeb-B	2	0.0	0.3	0.2	1.9	9.5	14.0	2,135.3
	4	0.0	0.4	0.4	6.5	5.8	7.5	2,793.1
	8	0.0	0.8	0.5	10.0	14.0	14.7	3,590.5
	16	0.2	1.5	1.1	23.7	24.9	21.2	3,720.0
ClueWeb-A	2	0.1	1.5	1.2	11.1	171.3	268.1	7,696.0
	4	0.1	2.2	1.8	21.5	26.1	25.3	8,965.3
	8	0.1	3.1	3.2	59.2	46.4	58.2	27,306.5
	16	0.2	6.2	4.9	100.8	86.3	121.1	13,212.2
YWC	2	3.7	151.8	133.4	1,231.4	33,438.0	68,740.4	–
	4	5.0	230.9	180.6	2,387.5	3,716.3	8,744.0	–
	8	7.3	312.0	280.6	5,078.4	2,393.8	5,129.7	–
	16	14.5	598.4	530.9	3,697.1	3,225.0	3,985.9	–

and Suff+ exhibit a slightly different behavior as the lowest running times are not achieved when $K = 2$. This can be explained by the variation in the ratio of faster MinMin+-based assignments.

5.6 Related Work on Web Crawling

A large body of literature exists on web crawling [9, 22, 26, 27, 30, 44, 83, 88, 106]. Some practical crawler designs are disclosed in [58, 73, 102, 109]. For a survey of literature on web crawling, interested reader may refer to [87]. A good overview of practical issues in parallel web crawling can be found in [25]. Herein, we omit the previous works on parallel web crawling [9, 25, 58, 102, 109], and focus mainly on geographically distributed web crawling [16, 17, 40, 41, 49].

In [17], where simulations are conducted to assess the performance benefits in crawling the Web from multiple, geographically distant data centers. Significant improvements are reported in page download throughput relative to the centralized web crawling scenario. However, [17] does not algorithmically treat the server-to-crawler assignment problem (the web servers are assigned to crawlers based on the servers' top-level country domains). The work in [49] formulated

the assignment problem as a geographically focused web crawling problem.

A few works considered reducing the communication cost of inter-crawler link exchange. In [18], a combinatorial model for assigning web pages to crawlers is proposed to minimize the amount of inter-crawler link exchange. This work, assumes that the Web is crawled from a central location, by crawlers running on the nodes of a large cluster. In [25], the communication overhead of link exchange is investigated for a parallel crawling setting. This work suggests exchanging the links in batch mode, at regular time intervals. In our work, we do not focus on this type of an overhead as we assume that the volume of link data exchanged between crawlers will be orders of magnitude lower relative to the volume of data downloaded from web servers.

Chapter 6

Independent Task Assignment of Very Large Sets: A Multi-Level Approach

In this chapter, we propose a multi-level task-to-processor assignment approach, which can yield good solutions in reasonable time even when the number of tasks in the dataset is very large. We will refer to this approach as `Multi-level` later on. As summarized in Section 2.3, multi-level paradigm has been successfully and widely used in solving the graph and hypergraph partitioning problems. The algorithms have three stages: coarsening, initial solution, and uncoarsening. Those algorithms use the connections (edges) of the underlying model especially in the coarsening and uncoarsening phases. However, in the independent task assignment problem, there is no underlying connections between tasks, thus most of the coarsening techniques used in the graph/hypergraph partitioning are not suitable for the independent task assignment. In this chapter, while we follow the main structure of coarsening-initial solution-uncoarsening phases in our multi-level algorithm, we propose novel techniques for those phases, designed especially for the independent task assignment problem on very large datasets.

Similar to its use in multi-level graph/hypergraph partitioning, our multi-level assignment approach involves three phases: task coarsening, initial task-to-processor assignment, and task uncoarsening. The key components in these phases are the task clustering metric and its efficient implementation used in the coarsening phase, the assignment heuristic used in the initial assignment phase, and the iterative refinement technique used in the uncoarsening phase. We describe each phase, emphasizing the above-mentioned components.

The rest of the chapter is organized as follows. In Section 6.1, we present a novel coarsening algorithm. In Section 6.2, we present techniques to find an initial solution for the coarsest assignment instance. In Section 6.3, we propose refinements for the uncoarsening phase of the multi-level algorithm. In Section 6.4, experimental setup and results are presented.

6.1 Coarsening

In the coarsening phase, the original task-to-processor assignment problem $(\mathcal{T}^0, X^0)$ is coarsened through a Z-level coarsening process into a sequence of smaller assignment instances $\langle(\mathcal{T}^1, X^1), (\mathcal{T}^2, , X^2), \ldots, (\mathcal{T}^Z, , X^Z)\rangle$ such that $|\mathcal{T}^0| \geq |\mathcal{T}^1| \geq \ldots \geq |\mathcal{T}^Z|$, where the processor set $\mathcal{P}$ remains unchanged. At each coarsening level z, the tasks in set $\mathcal{T}^{z-1}$ are clustered via matching heuristics. Every cluster of tasks in $\mathcal{T}^{z-1}$ forms a separate super-task in the coarser set $\mathcal{T}^z$. The number of coarsening levels, Z, is determined, on the fly, by the number of tasks in the current coarsest set, i.e., the coarsening stops when $|\mathcal{T}^Z|$ drops below a threshold (typically, around 1000).

At each coarsening level z, the matching heuristic performs multiple iterations over the tasks in $\mathcal{T}^{z-1}$. At each iteration, two tasks T_i and T_j are selected from $\mathcal{T}^{z-1}$ and clustered together, creating a new super-task T_{ij} in $\mathcal{T}^z$. T_{ij} effectively inherits all previous tasks clustered in T_i and T_j. In order for a solution of the coarse assignment instance $(\mathcal{T}^z, X^z)$ to be good with respect to that of $(\mathcal{T}^0, X^0)$, the ETC value $x_{ij,k}$ of the super-task T_{ij} is computed as the sum of the respective

ETC values of T_i and T_j, i.e.,

$$x_{ij,k} = x_{i,k} + x_{j,k} \text{ for } k = 1, \ldots, K. \tag{6.1}$$

This is required when we evaluate the quality of a solution of a coarser assignment instance to that of the original assignment instance. In this way, the makespan of the solution found for a coarser task-to-processor assignment instance will be equal to the makespan of the solution of the finer assignment instances including the original instance.

6.1.1 Dissimilarity Metric for Matching

The objective in the matching heuristic is to cluster tasks with similar execution features during the coarsening phase. When matching two tasks, we adopt an optimistic measure that computes the opportunity loss in the best-case execution times observed for the given matching. More specifically, given two tasks T_i and T_j, our measure computes the minimum possible execution times, separately, for the two cases where the tasks are matched or not matched. If the two tasks are individually assigned to their favorite processors without any matching, the minimum execution cost for the two tasks will be is equal to

$$\min_k x_{i,k} + \min_k x_{j,k}. \tag{6.2}$$

If the two tasks are matched into a super-task T_{ij}, however, the minimum execution cost for the two tasks becomes

$$\min_k \left(x_{i,k} + x_{j,k}\right) \tag{6.3}$$

The difference between the two cost values given in Eqs. 6.2 and 6.3 defines the following dissimilarity metric

$$\alpha_{ij} = \min_k \left(x_{i,k} + x_{j,k}\right) - \left(\min_k x_{i,k} + \min_k x_{j,k}\right). \tag{6.4}$$

α_{ij} is a dissimilarity metric because it shows the potential increase in execution times due to matching under the relaxed assumption that the tasks can always be

assigned to their favorite processors. Lower α_{ij} values imply favorite processors with a similar execution performance for tasks T_i and T_j. α_{ij} is always nonnegative and α_{ij} becomes zero when the favorite processors of T_i and T_j are the same.

6.1.2 Efficient Matching Algorithm

At each coarsening level, after we compute all possible α_{ij} values between the task pairs, the problem of finding the best matching can be formulated as the minimum-weight graph matching problem. There exists an optimal solution for this problem [71], but it is computationally very expensive. A faster but suboptimal solution is based on sorting all possible task pairs in increasing order of their α_{ij} values. Then, the pairs are visited in this order and the tasks in a pair are matched if neither of the tasks is matched before. In practice, even this solution is quite expensive since it requires $O(N^2 \log N)$ time.

In our work, we use an effective two-stage matching heuristic that runs in reasonable time. The main objective of the proposed algorithm is to avoid computing α_{ij} values between all task pairs which incurs $O(N^2)$ time. In the first stage, we compute the favorite κ processors of each task. A task is considered for clustering with only the tasks that have an identical κ-favorite-processor set. Note that the favorite-processor ordering information of the tasks are not preserved in these sets. That is, two tasks considered for matching may have different favorite-processor orderings. To efficiently compute the candidate processor sets, we generate buckets for κ-favorite-processor sets and place each task to its corresponding bucket. We utilize a trie data structure to effficiently access the corresponding bucket. While placing a task to a bucket, we consider matching task pairs whose favorite processors are identical. The buckets also contain a favorite-processor-to-task map to retrieve identical processors in $O(1)$ time. Note that these task pairs will have the minimum possible α values of zero. In the second stage, we use the above mentioned $O(N^2 \log N)$-time random-visit algorithm on the remaining unmatched tasks, where N is equal to the number r of tasks that remained unmatched after the first stage.

To prevent further clustering of tasks with larger weights, we use a filtering approach. In particular, if the minimum execution time of a task is larger than a threshold, the corresponding vertex is not considered for matching. After some empirical analyses, we decided to use the minimum execution time of the 250th largest task as the threshold value. This is especially useful if the dataset is skewed.

The proposed matching algorithm is illustrated in Algorithm 6.1, where Ψ is the set that maintains the matched tasks and U is the set of unassigned tasks. Ψ is also the output of the algorithm. These two sets are initialized at lines 1 and 2. The for loop at lines 4–18 visits each task T_i in random order for a possible matching. The threshold if-block at lines 5–8 prevents matching of largest tasks using the given size threshold. Line 9 finds the favorite κ processors of task T_i and line 10 determines the bucket for this κ-processor set using a trie data structure. Line 11 determines the favorite processor k of task T_i. The if-then-else block at lines 12–18 checks whether there exists a task T_j with the same favorite processor k of T_i in the same bucket. If such a task T_j is found, T_i and T_j are matched and T_j is removed from the bucket at lines 16–18. Otherwise task T_i remains unmatched in the current iteration and it is added to the bucket for a possible matching in the following iterations. Lines 20–26 calculate and sort the α_{ij} values for the remaining unmatched tasks. Lines 27–30 traverse the α_{ij} pairs in ascending order and try to match the unmatched tasks.

The running time analysis of the matching algorithm for a coarsening level that contains $N_z = |S^z|$ tasks is as follows: The threshold if-block (lines 5–8) executes in $O(K)$ time. Line 9 also executes in $O(K)$ time. Using the trie data structure, line 10 runs in $O(\kappa \log \kappa)$ time. Finding the favorite processor takes $O(K)$ time. Utilizing the favorite-processor-to-task map, line 12 runs in $O(1)$ time. The rest of the loop runs in $O(1)$ time. An iteration of the for loop at lines 4–18 runs in $O(K + \kappa \log \kappa)$ time. So the first stage of the algorithm runs in $O(KN_z + \kappa N_z \log \kappa)$ time.

The second stage executes in $O(r^2 \log r)$ time, where r is the number of unmatched tasks after the first stage. So, the running time of the overall algorithm

Algorithm 6.1 MATCHING(X, K, N, *threshold*, κ)

```
1:  Ψ ← ∅                                                    ▷ Ψ: set of matched tasks
2:  U ← {1, 2, ..., N}
3:  ▷ First stage:
4:  for each i ∈ {1, ..., N} in random order do
5:      if min_k{x_{i,k}} > threshold then                   ▷ large task, do not cluster
6:          U ← U − {i}
7:          Ψ ← Ψ ∪ {⟨i⟩}
8:          continue loop i
9:      processorSet ← FavoriteProcessors(i, K, κ)
10:     bucket ← BucketOf(processorSet )
11:     k ← argmin_k x_{i,k}
12:     if (j ← TaskWithFavoriteProcessor(bucket, k)) is nil then
13:                       ▷ j is the task with favorite processor P_k in the bucket, if any
14:         AddToBucket(bucket, i, k)
15:     else
16:         Ψ ← Ψ ∪ {⟨i, j⟩}
17:         U ← U − {i, j}
18:         RemoveFromBucket(bucket, j)
19: ▷ Second stage:
20: D ← ∅
21: for each i ∈ U do
22:     for each j ∈ U do
23:         if i ≠ j then
24:             α_ij ← min_k (x_{i,k} + x_{j,k}) − (min_k x_{i,k} + min_k x_{j,k})
25:             D ← D ∪ {⟨i, j, α_ij⟩}
26: Sort(D)                                      ▷ sort D array by ascending α_ij values
27: for each ⟨i, j, α_ij⟩ ∈ D do
28:     if i ∈ U and j ∈ U then                                        ▷ not matched yet
29:         Ψ ← Ψ ∪ {⟨i, j⟩}
30:         U ← U ∪ {i, j}
31: return Ψ
```

is $O(N_zK+\kappa N_z \log\kappa + r^2 \log r)$. The following proofs discuss the total execution time of the coarsening phase.

Lemma 6.1.1 *The number r of tasks that remained unmatched after the first stage is at most $\kappa\binom{K}{\kappa}$.*

Proof: In a bucket, there can be at most κ unmatched tasks, because of the strategy of matching tasks with identical favorite processors in a bucket. There are at most $\binom{K}{\kappa}$ buckets, and thus the number r of tasks that remained unmatched after the first stage is at most $\kappa\binom{K}{\kappa}$. ■

Lemma 6.1.2 *For $\kappa = 2$, the running time of a coarsening level z is $O(N_zK + K^4 \log K)$.*

Proof: For $\kappa = 2$, the running time of a coarsening level z is

$$O(N_zK + \kappa N_z \log\kappa + r^2 \log r). \tag{6.5}$$

Given $\kappa = 2$ and the constraint on r,

$$r \leq \kappa\binom{K}{\kappa} = K^2 - K, \tag{6.6}$$

the running time of a coarsening level z becomes $O(N_zK + K^4 \log K)$. ■

Theorem 6.1.1 *The running time of the total coarsening phase is $O(NK + K^4 \log K \log N)$ for $\kappa = 2$.*

Proof: Only very few large tasks are excluded from matching and all remaining tasks (expect possibly one) are matched. Hence, we can assume that the number of tasks reduce by a factor of two at each coarsening level. Thus, the number of levels is at most $O(\log N)$ and the running time of the total coarsening phase becomes

$$O\left(\sum_{z=0}^{\log N}\left(K\frac{N}{2^z} + K^4 \log K\right)\right) = O(NK + K^4 \log K \log N), \tag{6.7}$$

for $\kappa = 2$. ■

Note that, if $N/\log N = \Omega(K^3 \log K)$ then running time of the total coarsening phase becomes $O(NK)$ for $\kappa = 2$.

6.2 Initial Task-to-Processor Assignment

In this phase, we obtain an initial task-to-processor assignment by running one of the constructive heuristics on the coarsest assignment instance $(\mathcal{T}^Z, X^Z)$. We experimented with constructive assignment heuristics for this phase, and `MinMin+` appeared to be the best choice, even for the skewed datasets. `MaxMin+` and `Suff+` are executing much better on the skewed datasets as we report in Chapter 4. The multi-level task clustering approach adopted in the coarsening phase have the tendency to balance the execution costs of super-tasks, thus decreasing the skewness of the assignment instances as the coarsening proceeds, leading to a most-probably non-skewed assignment instance at the coarsest level. In Chapter 4, `MinMin+` is already found to perform better than `MaxMin+` and `Suff+` in the independent task assignment of non-skewed datasets. This reason stands as an explanation of better performance of `MinMin+` compared to `MaxMin+` and `Suff+` as an initial assignment algorithm even for the skewed datasets.

6.3 Uncoarsening

During the uncoarsening phase, the solution A^Z found in the initial assignment phase for the coarsest assignment instance $(\mathcal{T}^Z, X^Z)$ is projected back to a solution A^0 for the original assignment instance $(\mathcal{T}^0, X^0)$ by going through the finer instances $(\mathcal{T}^{Z-1}, X^{Z-1}), \ldots, (\mathcal{T}^0, X^0)$. That is, at each level z of the uncoarsening phase, the task-to-processor assignment A^{z+1} found for $\mathcal{T}^{z+1}$ is projected back to a task-to-processor assignment A^z for $\mathcal{T}^z$. In this projection, the constituent tasks S_i^z and S_j^z of each super task S_{ij}^{z+1} are both assigned to the same processor to which S_{ij}^{z+1} is assigned, i.e., $A^z[i] = A^z[j] = A^{z+1}[ij]$. Because of the

clustering scheme adopted in the coarsening phase (see Eq. 6.4), the assignment A^z obtained by the projection has the same makespan with the assignment A^{z+1}. Fortunately, A^z can be improved further in the finer level because of the higher degree of freedom in the task-to-processor assignment of the finer level compared to the coarser level. That is, for a super task S_{ij}^{z+1}, S_i^z and S_j^z are restricted to be assigned to the same processor at the coarser level $z+1$, whereas they have the flexibility of being assigned to different processors at the finer level z.

The task-to-processor assignment obtained by the projection is improved through a number of refinement passes, each involving a sequence of reassignments of tasks to processors. We will describe our proposed refinement methods in the following subsections.

6.3.1 Move Refinement

Move refinement is a refinement method where at each transaction a single change in the assignment of a single task is considered, i.e., we *move* a single task from one processor to another. At the beginning of each refinement pass, for each processor P_k, a task reassignment list is obtained by sorting the tasks assigned to P_k in decreasing order of their execution costs with respect to P_k. For each of the reassignment lists, we maintain a pointer that initially shows the first task in the list. The tasks of the bottleneck processor are traversed in the given order for reassignment. This reassignment scheme is adopted because a possible reassignment of a task with the maximum execution time to a bottleneck processor (P_b) is likely to decrease the makespan more than the others. For each task in the list, we consider all reassignments that do not increase the makespan and, for those target processors, we compute the reassignment gains. Then, we choose the one with the maximum gain. If the maximum reassignment gain of the current task to any other processor is positive, then we reassign the task to the target processor and advance the pointer for the list of P_b. If the gain is not positive, we skip the current task by advancing the pointer anyway. A reassigned task is not included in the reassignment list of the target processor. Hence, this task is not considered for further reassignments in the current pass. A task reassignment

may also change the bottleneck processor P_b. If bottleneck processor changes, the traversal continues on the list of the new bottleneck processor starting from the respective pointer. A refinement pass ends when the last task of the current bottleneck processor is processed for reassignment. In an uncoarsening level, the refinement passes are terminated until there is no change in the assignments in a pass.

The task reassignment gains are computed based on the decrease in the makespan due to the reassignment. That is, the gain of assigning to a processor P_k a task T_i that is already assigned to a bottleneck processor P_b is computed as

$$g_i(b \to k) = e_b - \max\{e_b - x_{i,b}, e_k + x_{i,k}\}. \tag{6.8}$$

Here, e_b and the max term denote the maximum load of processors P_b and P_k before and after the reassignment, respectively.

A single pass of the refinement step is illustrated in Algorithm 6.2. Here, SRL_k represents the task reassignment list of processor P_k. Each p_k value denotes the pointer maintained for the corresponding task reallocation list SRL_k. s_k denotes the number of tasks initially assigned to processor P_k. The first two loops (lines 1–9) initialize and calculate the p_k, s_k, and SRL_k variables. The third loop (lines 10–11) sorts the SRL_k arrays in decreasing order of $x_{i,k}$ values. Line 12 determines the bottleneck processor. The while-loop traverses the task reassignment list of the bottleneck processor and stops when the list of the bottleneck processor is exhausted. The first for-loop inside the while-loop considers all processor alternatives and tries to find one with a positive gain. If a processor with a positive gain is found, lines 23–26 perform the reassignment. In this algorithm, the gain of an individual task reassignment can be computed in $O(K)$ time. Since at most N_z reassignments are considered in a single pass, the running time of a single reassignment pass is $O(N_zK)$ at level z of the uncoarsening phase.

Algorithm 6.2 MOVEREFINEMENT(X, K, N, A)

```
1: for k ← 1 to K do
2:     p_k ← 1
3:     s_k ← 0
4:     e_k ← 0
5: for i ← 1 to N do
6:     k ← A[i]
7:     s_k ← s_k + 1
8:     SRL_{k,s_k} ← i
9:     e_k ← e_k + x_{i,k}
10: for k ← 1 to K do
11:     SORT(SRL_k)            ▷ sort SRL_k array by decreasing x_{i,k} values
12: b ← argmax_k e_k
13: while p_b ≤ s_b do
14:     i ← SRL_{b,p_b}
15:     maxg ← −∞
16:     time_gain ← −∞
17:     for k ← 1 to K do
18:         g ← e_b − max{e_b − x_{i,b}, e_k + x_{i,k}}
19:         if g > maxg or (g = maxg and time_gain < x_{i,k} − x_{i,b}) then
20:             maxg ← g
21:             time_gain ← x_{i,k} − x_{i,b}
22:             kmax ← k
23:     if maxg > 0 or (maxg = 0 and time_gain > 0) then
24:         A[i] ← kmax
25:         e_b ← e_b − x_{i,b}
26:         e_kmax ← e_kmax + x_{i,kmax}
27:     p_b ← p_b + 1
28:     b ← argmax_k e_k
```

6.3.2 Swap Refinement

Move refinement works well when there is room to move tasks between processors. However, if all processors are balanced then move operations are not allowed since all moves will increase bottleneck values. This is the case even if the move of the task drastically decreases total execution time. On the other hand, swap operations, which is the exchange of assigned processors for a selected pair of tasks, may still refine the assignments.

The swap reassignment gains are computed based on the decrease in the makespan due to the reassignment. The gain of exchanging a task T_i assigned to a bottleneck processor P_b, with a task T_j assigned to a processor P_ℓ is computed as

$$g_{ij}(b \leftrightarrow \ell) = e_b - \max\{e_b - x_{i,b} + x_{j,b}, e_\ell - xj, \ell + x_{i,\ell}\}. \tag{6.9}$$

If we consider each possible pair at each swap operation, the algorithm would run in $O(N_z^3)$ time: $O(N_z^2)$ time is to traverse every possible pair and the operation is repeated after each successful swap. This is quite impractical except for the smallest values of N_z. We thus investigate better approaches to consider candidates of the swap operation.

The approach of move refinement to traverse candidate tasks will also be our base for the swap refinement. Similar to the move refinement, we traverse the tasks of the bottleneck processor in decreasing order, and swap to the new bottleneck processor when bottleneck changes. We do not reconsider a task again for a swap operation if it is already chosen for exchange in the current pass. We will traverse $O(N_z)$ tasks in a single pass. To find the best pair for this task, we may perform a brute force algorithm, which examines every possible pair. The resulting algorithm will be $O(N_z^2)$ at level z of the uncoarsening phase. This algorithm is faster than $O(N_z^3)$, which can be utilized for larger values of N_z, but it is still impractical on levels where we have hundreds of millions of tasks.

We investigate further whether there are more practical implementations of selecting the best swap pair of the task T_i. We will describe an approach utilizing

a two-dimensional query structure.

Remember that we are to find a best matching pair task T_j for a task T_i which is currently assigned to a bottleneck processor P_b. Let P_ℓ is a processor we want to swap a task with. We want to find a task T_j among the tasks assigned to processor P_b which generates a best swap gain.

The swap operation has a positive gain on P_b when $x_{i,b} > x_{j,b}$. The swap has a positive gain on P_ℓ when $e_\ell - x_{j,\ell} + x_{i,\ell} < e_b$. The best task T_j can be determined as

$$\begin{aligned}
\operatorname*{argmax}_j \{g_{ij}(b \leftrightarrow \ell)\} &= \operatorname*{argmax}_{\{j|A_j=\ell\}} \{e_b - \max\{e_b - x_{i,b} + x_{j,b}, e_\ell - x_{j,\ell} + x_{i,\ell}\}\} \\
&= \operatorname*{argmax}_{\{j|A_j=\ell\}} \{-\max\{e_b - x_{i,b} + x_{j,b}, e_\ell - x_{j,\ell} + x_{i,\ell}\}\} \\
&= \operatorname*{argmin}_{\{j|A_j=\ell\}} \{\max\{e_b - x_{i,b} + x_{j,b}, e_\ell - x_{j,\ell} + x_{i,\ell}\}\}
\end{aligned}$$

Let $\gamma_{i,b} = e_b - x_{i,b}$ and $\beta_{i,\ell} = e_\ell + x_{i,\ell}$. The gain equation becomes

$$\operatorname*{argmax}_{\{j|A_j=\ell\}} \{g_{ij}(b \leftrightarrow \ell)\} = \operatorname*{argmin}_{\{j|A_j=\ell\}} \{\max\{\alpha_{i,b} + x_{j,b}, \beta_{i,\ell} - x_{j,\ell}\}\}$$

To better analyze the internal max operation, we split the task data into two. Among the tasks T_j with $\gamma_{i,b} + x_{j,b} \geq \beta_{i,\ell} - x_{j,\ell}$,

$$\operatorname*{argmax}_j \{g_{ij}(b \leftrightarrow \ell)\} = \operatorname*{argmin}_j \{\gamma_{i,b} + x_{j,b}\} = \operatorname*{argmin}_j \{x_{j,b}\} \tag{6.10}$$

and we are to find a minimum $x_{j,b}$ for those tasks. Similarly, among the tasks T_j with $\gamma_{i,b} + x_{j,b} < \beta_{i,\ell} - x_{j,\ell}$,

$$\operatorname*{argmax}_j \{g_{ij}(b \leftrightarrow \ell)\} = \operatorname*{argmin}_j \{\beta_{i,\ell} - x_{j,\ell}\} = \operatorname*{argmax}_j \{x_{j,\ell}\} \tag{6.11}$$

and we are to find a maximum $x_{j,\ell}$ for those tasks.

To better utilize the two-dimensional query mechanism, we define three functions:

$$\begin{aligned}
f^1_{\ell,b}(j) &= x_{j,b} + x_{j,\ell} \\
f^2_{\ell,b}(j) &= x_{j,b} \\
f^3_{\ell,b}(j) &= x_{j,\ell}
\end{aligned}$$

We populate two two-dimensional systems, Q_1 with $\langle x, y\rangle = \langle f^1_{\ell,b}(j), f^2_{\ell,b}(j)\rangle$ tuples and Q_2 with $\langle x, y\rangle = \langle f^1_{\ell,b}(j), f^3_{\ell,b}(j)\rangle$ tuples. To find the task with maximum gain, we are to perform the following queries on Q_1 and Q_2.

On Q_1, we perform a query to find the task T_j with minimum y constrainted by $x \geq \beta_{i,l} - \gamma_{i,b}$. Similarly, on Q_2, we perform a query to find the task T_j with maximum y constrainted by $x < \beta_{i,l} - \gamma_{i,b}$. We then compare these two tasks to determine the winner.

For a task T_i on the bottleneck processor P_b, we perform the above calculation for every other processor to choose their candidate swap tasks, we then compare the results and determine the task T_j for the swap operation.

We generate these two-dimensional queries for each P_k and P_ℓ processor pairs at the beginning of a swap reassignment pass. We implemented the query mechanism on a kd-tree. A kd-tree can be initialized in $O(N \log N)$ time. Thus the initialize of $O(K^2)$ pairs of kd-trees will be in $O(K^2 N_z \log N_z)$ time at level z of the uncoarsening phase. The average query operation on a kd-tree is $O(\log N)$ for an input of size N. The selection of a pair will take $O(K \log N_z)$ average time, which will perform $O(N_z)$ times, thus accumulate to $O(K N_z \log N_z)$ average time. Hence, a single pass of the uncoarsening level z can be performed in $O(K^2 N_z \log N_z)$ average time. Since $K \ll N_z$, this complexity is better than to the previous candidate of $O(N_z^2)$.

A single pass of the swap refinement step is illustrated in Algorithm 6.3. SRL_k, p_k and s_k values are used with the same purpose as in Algorithm 6.2. The first two loops (lines 1–9) initialize and calculate the p_k, s_k, and SRL_k variables. The third loop (lines 10–11) sorts the SRL_k arrays in decreasing order of $x_{i,k}$ values, as in Algorithm 6.2. Line 12 initializes the $2K^2$ kd-trees using Algorithm 6.4. Line 12 determines the bottleneck processor. The while-loop traverses the task reassignment list of the bottleneck processor and stops when the list of the bottleneck processor is exhausted. The first for-loop inside the while-loop considers all processor alternatives and tries to find one with a positive gain. Algorithm 6.5 is utilized to determine the candidate swap pair for the interested task on the bottleneck processor (line 16). If a processor with a positive gain is found, lines 23–26

Table 6.1: Properties of the datasets

Dataset	# of tasks
ClueWeb-B	0.8 million
ClueWeb-A	2.5 million
YWC	137.7 million

perform the reassignments.

6.4 Experiments

We conduct our simulations on three different web datasets: ClueWeb-B, ClueWeb-A, and YWC. These datasets are the datasets we used in Chapter 5. The properties of the three datasets are displayed in Table 6.1. In all datasets, we observe a highly skewed task distribution, i.e., there are small tasks, but few very large tasks.

We compare our algorithm against the well-known heuristics MET, PPB, MCT, MinMin+, MaxMin+, Suff+ and GA+. Other algorithms known to produce good results such as RC [107], RASA [90] are not included in our comparison because of their high running time complexities. The heuristics are implemented in the Java programming language. The simulations are carried out on a Linux workstation equipped with six 2100-MHz quad-core CPUs and 132 GB of memory. Since the MCT, PPB, GA+ and Multi-level heuristics involve non-deterministic components, these heuristics were executed 10 times with different random seeds for each assignment instance and the average performance figures are reported. Due to the high running time and extremely high memory requirements of the GA+ heuristic, it was not feasible to complete the simulations on the YWC dataset for this heuristic. The incomplete results are indicated by "–" in the following tables.

The most important performance metric in independent task assignment problem is the makespan value. We define the load imbalance %LI of an assignment

Algorithm 6.3 SwapRefinement(X, K, N, A)

```
1: for k ← 1 to K do
2:     p_k ← 1
3:     s_k ← 0
4:     e_k ← 0
5: for i ← 1 to N do
6:     k ← A[i]
7:     s_k ← s_k + 1
8:     SRL_{k,s_k} ← i
9:     e_k ← e_k + x_{i,k}
10: for k ← 1 to K do
11:     Sort(SRL_k)          ▷ sort SRL_k array by decreasing x_{i,k} values
12: Q ← GenerateKdTrees(X, K, N, A)
13: b ← argmax_k e_k
14: while p_b ≤ s_b do
15:     i ← SRL_{b,p_b}
16:     ⟨j, maxg, time_gain⟩ ← FindBestSwapPair(i, A, X, e)
17:     if maxg > 0 or (maxg = 0 and time_gain > 0) then
18:         ℓ ← A[j]
19:         A[i] ← ℓ
20:         A[j] ← b
21:         e_b ← e_b − x_{i,b} + x_{j,b}
22:         e_ℓ ← e_ℓ − x_{j,ℓ} + x_{i,ℓ}
23:         for k ← 1 to K do
24:             DeleteKdTree(Q_{k,ℓ,1}, j)
25:             DeleteKdTree(Q_{k,ℓ,2}, j)
26:     p_b ← p_b + 1
27:     b ← argmax_k e_k
```

Algorithm 6.4 GenerateKdTrees(X, K, N, A)

```
 1: for k ← 1 to K do
 2:     for ℓ ← 1 to K do
 3:         if k = ℓ then
 4:             continue loop ℓ
 5:         A_ℓ ← AssignedTasks(ℓ, A, N)
 6:                                   ▷ Set of assigned tasks to processor P_ℓ
 7:         B_1 ← EmptyList()
 8:         B_2 ← EmptyList()
 9:         for each i ∈ A_ℓ do
10:             b_1.key ← i
11:             b_1.x ← x_{i,k}
12:             b_1.y ← x_{i,k} + x_{i,ℓ}
13:             AppendTo(b_1, B_1)
14:             b_2.key ← i
15:             b_2.x ← x_{i,ℓ}
16:             b_2.y ← x_{i,k} + x_{i,ℓ}
17:             AppendTo(b_2, B_2)
18:         Q_{k,ℓ,1} ← InitializeKdTree(B_1)
19:         Q_{k,ℓ,2} ← InitializeKdTree(B_2)
20:            ▷ initialize the kd-trees using the populated lists B_1 and B_2
21: return Q
```

Algorithm 6.5 FINDBESTSWAPPAIR(i, A, X, e)

1: $k \leftarrow A_i$
2: $\gamma \leftarrow e_k - x_{i,k}$
3: **for** $\ell \leftarrow 1$ **to** K **do**
4:     **if** $k = \ell$ **then**
5:         **continue** loop ℓ
6:     $\beta \leftarrow e_\ell + x_{i,\ell}$
7:     $z \leftarrow \beta - \gamma$
8:     $j \leftarrow$ PICKMINIMUMX($Q_{k,\ell,1}$, $y \geq z$)
9:     $g \leftarrow e_k - \max\{e_k - x_{i,k} + x_{j,k}, e_\ell + x_{i,\ell} - x_{j,\ell}\}$
10:     **if** $g > maxg$ **or** ($g = maxg$ **and** $time_gain < x_{i,k} - x_{i,\ell} + x_{j,\ell} - x_{j,k}$) **then**
11:         $maxg \leftarrow g$
12:         $time_gain \leftarrow x_{i,k} - x_{i,\ell} + x_{j,\ell} - x_{j,k}$
13:         $jmax \leftarrow j$
14:     $j \leftarrow$ PICKMAXIMUMX($Q_{k,\ell,2}$, $y < z$)
15:     $g \leftarrow e_k - \max\{e_k - x_{i,k} + x_{j,k}, e_\ell + x_{i,\ell} - x_{j,\ell}\}$
16:     **if** $g > maxg$ **or** ($g = maxg$ **and** $time_gain < x_{i,k} - x_{i,\ell} + x_{j,\ell} - x_{j,k}$) **then**
17:         $maxg \leftarrow g$
18:         $time_gain \leftarrow x_{i,k} - x_{i,\ell} + x_{j,\ell} - x_{j,k}$
19:         $jmax \leftarrow j$
20: **return** $\langle jmax, maxg, time_gain \rangle$

Table 6.2: Percent load imbalance values

Test	K	MET	PPB	MCT	MinMin+	MaxMin+	Suff+	GA+	Multi-level
ClueWeb-B	2	36.42	36.42	26.20	16.93	13.46	14.38	14.42	**3.78**
	4	48.69	48.45	48.34	63.90	18.16	22.60	60.01	**17.78**
	8	330.56	164.43	166.60	208.47	103.22	104.62	199.73	**89.62**
	16	794.61	384.27	361.94	420.30	**244.72**	**244.72**	410.51	246.13
ClueWeb-A	2	49.64	49.64	44.26	16.80	21.05	21.02	**15.97**	18.76
	4	32.77	22.29	53.80	16.02	11.53	11.31	14.40	**8.83**
	8	216.18	79.36	86.02	76.81	58.32	61.19	74.32	**56.74**
	16	663.36	169.77	152.95	171.83	128.95	127.51	168.50	**122.50**
YWC	2	6.33	6.33	30.87	1.24	0.47	1.34	–	**0.27**
	4	23.09	27.01	53.01	7.58	5.46	5.80	–	**5.38**
	8	97.33	55.59	67.70	23.91	18.77	**17.76**	–	21.79
	16	367.85	110.73	110.71	96.74	**68.98**	69.54	–	69.21

heuristic relative to the ideal makespan as

$$\%\text{LI} = \frac{M - M^*}{M^*}, \tag{6.12}$$

where M denotes the makespan of the assignment produced by the heuristic and M^* denotes the ideal makespan for the given assignment instance. The ideal makespan is computed as

$$M^* = \frac{W^*_{\text{tot}}}{K} = \frac{\sum_i \min_k\{x_{i,k}\}}{K}, \tag{6.13}$$

where W^*_{tot} is a lower bound on the makespan that would be attained if each task was assigned to its favorite processor. Note that the M^* value is a rather loose lower bound on the makespan, i.e., the optimal makespan is very likely to be greater than M^*.

Table 6.2 displays the load imbalance values for the 2-, 4-, 8-, and 16-way assignments produced by the evaluated task assignment heuristics. The bold values in each row indicate the best performing heuristic(s) for the corresponding assignment instance. As seen in Table 6.2, the fast assignment heuristics `MET`, `PPB`, `MCT` produces low quality solutions. `MinMin+` seems better than those heuristics, but fails to find better solutions in some of the assignment instances especially for larger K. `MaxMin+` and `Suff+` produces comparable results, where `MaxMin+` is a step ahead. `GA+` produces better solutions than `MinMin+`, and succeeds to find better solutions than `MaxMin+` and `Suff+` in some assignment instances, but fails to find a better solution in the majority of the instances. Moreover, `GA+` could

Table 6.3: Execution times (in seconds)

Test	K	MET	PPB	MCT	MinMin+	MaxMin+	Suff+	GA+	Multi-level
ClueWeb-B	2	0.0	0.3	0.2	1.9	9.5	14.0	2,135.3	79.5
	4	0.0	0.4	0.4	6.5	5.8	7.5	2,793.1	120.9
	8	0.0	0.8	0.5	10.0	14.0	14.7	3,590.5	719.5
	16	0.2	1.5	1.1	23.7	24.9	21.2	3,720.0	266.5
ClueWeb-A	2	0.1	1.5	1.2	11.1	171.3	268.1	7,696.0	62.1
	4	0.1	2.2	1.8	21.5	26.1	25.3	8,965.3	115.8
	8	0.1	3.1	3.2	59.2	46.4	58.2	27,306.5	297.5
	16	0.2	6.2	4.9	100.8	86.3	121.1	13,212.2	576.5
YWC	2	3.7	151.8	133.4	1,231.4	33,438.0	68,740.4	–	386.3
	4	5.0	230.9	180.6	2,387.5	3,716.3	8,744.0	–	631.5
	8	7.3	312.0	280.6	5,078.4	2,393.8	5,129.7	–	1,062.6
	16	14.5	598.4	530.9	3,697.1	3,225.0	3,985.9	–	2,008.6

not execute in our largest dataset YWC. Thus, we conclude GA+ is not appropriate to execute on large datasets.

Multi-level is best among all of those heuristics: it succeeds to find best solution in 8 assignment instances out of 12. MaxMin+ found two best solutions, Suff+ found two best solutions, where MaxMin+ and Suff+ share the lead for ClueWeb-A on $K = 16$ assignment instance. GA+ succeeds to find the best solution for ClueWeb-A $K = 2$.

Table 6.3 displays the running times of the heuristics for different types of datasets. The running times of MinMin+ generally increases with increasing K, as expected. The running times of MaxMin+ and Suff+ decreases significantly with increasing K for small values of K, then increases with increasing K for larger values of K. This is expected since increasing K values increases the ratio of faster MinMin+ based assignments. MaxMin+ and Suff+ algorithms require large preprocessing times, especially with larger datasets and smaller crawler counts. They require several hours to complete for the YWC dataset. Multi-level algorithm does not execute fastest for smaller ClueWeb-B and ClueWeb-B datasets, but it succeeds to find the solution within minutes. When the number of tasks get larger, Multi-level algorithm is 1-2 orders of magnitute faster than Suff+ and MaxMin+ heuristics, on the YWC dataset. Furthermore, Multi-level algorithm is 4-5 times faster than efficient MinMin+ algorithm on this largest dataset.

These results reveal the efficiency of `Multi-level` algorithm: When the number of tasks is very large, the use `Multi-level` algorithm is highly recommended.

Chapter 7

Conclusion

We studied the problem of one-dimensional partitioning of nonuniform workload arrays with optimal load balancing for heterogeneous systems. Within this study, we investigated two cases: chain-on-chain partitioning, where a chain of tasks is partitioned onto a chain of processors; and chain partitioning, where the task chain is partitioned onto a set of processors (i.e., permutation of the processors is allowed). We showed that chain-on-chain partitioning algorithms for homogenous systems can be revised to solve this partitioning problem for heterogeneous systems, without altering computational complexities of these algorithms. We proved that the chain partitioning problem is NP-complete, and empirically showed that exact CCP algorithms can serve as an effective heuristic, for the CP problem. Our experiments proved the effectiveness of our techniques, as the exact algorithms work much better than heuristics, and balanced work decompositions can be achieved even for high numbers of processors.

We presented certain performance improvements over the popular independent task assignment heuristics `MinMin`, `MaxMin`, and `Suff`. In particular, we proposed the `MinMin+` heuristic which improves the worst-case runtime complexity of `MinMin` from $O(KN^2)$ to $O(KN \log N)$ in assigning N independent tasks to K processors. Moreover, we proposed the `MaxMin+` and `Suff+` heuristics, which are hybrid versions of `MaxMin` and `Suff`, obtained by combining the latter heuristics with `MinMin`. We evaluated the performance of all heuristics over a large

number of real-life datasets. The experiments indicate that each of our heuristics runs considerably faster than their traditional counterparts, `MinMin+` being the fastest. In terms of the solution quality, both `MaxMin+` and `Suff+` are found to perform considerably better than `MinMin+` for skewed datasets while `MinMin+` is found to perform comparable for non-skewed datasets. Considering the tradeoffs between the solution quality and the running times of the proposed assignment algorithms, we recommend the use of `MinMin+` for non-skewed datasets and recommend `MaxMin+` for skewed datasets.

We adapted multi-level framework to the independent task assignment problem. We presented novel algorithms for the coarsening and uncoarsening phases for the proposed multilevel algorithm. We experimented the multilevel algorithm on very large task assignment problem instances. The results reveal that the performance of the proposed multilevel algorithm supersedes compared algorithms, both in quality and in runtime performance.

We demonstrated the improved solutions to the independent task assignment problem on distributed web crawling. We show that the assignment problem of distributed web crawling can be formulated as a task assignment problem. Our simulations on real-life web data collections and network statistics indicate that significant performance improvements can be attained by using independent task assignment algorithms.

Bibliography

[1] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Sahra Ali, and Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proceedings of the 9th IEEE Heterogeneous Computing Workshop*, HCW 2000, pages 185–199, Washington, DC, USA, 2000. IEEE Computer Society. doi:10.1109/HCW.2000.843743.

[2] Charles J. Alpert and Andrew B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration: The VLSI Journal*, 19(12):1–81, 1995. doi:10.1016/0167-9260(95)00008-4.

[3] S. Anily and A. Federgruen. Structured partitioning problems. *Operations Research*, 39(1):130–149, 1991. doi:10.1287/opre.39.1.130.

[4] Apache Software Foundation. Apache derby. `http://db.apache.org/derby/index.html`. Accessed: 2013-08-23.

[5] Robert Armstrong, Debra Hensgen, and Taylor Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proceedings of the 7th IEEE Heterogeneous Computing Workshop*, pages 79–87, Washington, DC, USA, 1998. IEEE Computer Society. doi:10.1109/HCW.1998.666547.

[6] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vassilis Plachouras, and Luca Telloli. On the feasibility of multi-site web search engines. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 425–434, New York, NY, USA, 2009. ACM. doi:10.1145/1645953.1646009.

[7] Roi Blanco, Berkant Barla Cambazoglu, Flavio P. Junqueira, Ivan Kelly, and Vincent Leroy. Assigning documents to master sites in distributed search. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 67–76, New York, NY, USA, 2011. ACM. doi:10.1145/2063576.2063591.

[8] Shahid H. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, 1988. doi:10.1109/12.75137.

[9] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: a scalable fully distributed Web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004. doi:10.1002/spe.587.

[10] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001. doi:10.1006/jpdc.2000.1714.

[11] Ulf Brefeld, Berkant Barla Cambazoglu, and Flavio P. Junqueira. Document Assignment in Multi-site Search Engines. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 575–584, New York, NY, USA, 2011. ACM. doi:10.1145/1935826.1935907.

[12] J. Bruno, E. G. Coffman, Jr., and R. Sethi. Scheduling Independent Tasks To Reduce Mean Finishing Time. *Communications of the ACM*, 17(7):382–387, July 1974. doi:10.1145/361011.361064.

[13] Thang Nguyen Bui and Curt Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452, Portsmouth, Virginia, USA, 1993. SIAM.

[14] Berkant Barla Cambazoglu and Cevdet Aykanat. Hypergraph-Partitioning-Based Remapping Models for Image-Space-Parallel Direct Volume Rendering of Unstructured Grids. *IEEE Transactions on Parallel and Distributed Systems*, 18(1):3–16, 2007. doi:10.1109/TPDS.2007.253277.

[15] Berkant Barla Cambazoglu and Ricardo Baeza-Yates. Scalability Challenges in Web Search Engines. In Massimo Melucci, Ricardo Baeza-Yates, and W. Bruce Croft, editors, *Advanced Topics in Information Retrieval*, volume 33 of *The Information Retrieval Series*, pages 27–50. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-20946-8_2.

[16] Berkant Barla Cambazoglu, Vassilis Plachouras, and Ricardo Baeza-Yates. Quantifying Performance and Quality Gains in Distributed Web Search Engines. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 411–418, New York, NY, USA, 2009. ACM. doi:10.1145/1571941.1572013.

[17] Berkant Barla Cambazoglu, Vassilis Plachouras, Flavio Junqueira, and Luca Telloli. On the Feasibility of Geographically Distributed Web Crawling. In *Proceedings of the 3rd International Conference on Scalable Information Systems*, InfoScale '08, pages 31:1–31:10, Brussels, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[18] Berkant Barla Cambazoglu, Ata Turk, and Cevdet Aykanat. Data-Parallel Web Crawling Models. In Cevdet Aykanat, Tugrul Dayar, and Ibrahim Korpeoglu, editors, *Proceedings of the 19th International Symposium on Computer and Information Sciences - ISCIS 2004*, volume 3280 of *Lecture Notes in Computer Science*, pages 801–809. Springer Berlin / Heidelberg, 2004. doi:10.1007/978-3-540-30182-0_80.

[19] Berkant Barla Cambazoglu, Emre Varol, Enver Kayaaslan, Cevdet Aykanat, and Ricardo Baeza-Yates. Query Forwarding in Geographically Distributed Search Engines. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 90–97, New York, NY, USA, 2010. ACM.

doi:10.1145/1835449.1835467.

[20] Umit V. Catalyurek and Cevdet Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999. doi:10.1109/71.780863.

[21] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert T. Heaphy, and Lee Ann Riesen. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, IPDPS'07, pages 1–11, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/IPDPS.2007.370258. Best Algorithms Paper Award.

[22] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11-16):1623–1640, 1999. doi:10.1016/S1389-1286(99)00052-3.

[23] Sameer Singh Chauhan and R. C. Joshi. QoS Guided Heuristic Algorithms for Grid Task Scheduling. *International Journal of Computer Applications*, 2(9):24–31, 2010. doi:10.5120/694-975.

[24] Hao Chen, Nicholas S. Flann, and Daniel W. Watson. Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):126–136, 1998. doi:10.1109/71.663870.

[25] Junghoo Cho and Hector Garcia-Molina. Parallel Crawlers. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 124–135, New York, NY, USA, 2002. ACM. doi:10.1145/511446.511464.

[26] Junghoo Cho and Hector Garcia-Molina. Effective Page Refresh Policies for Web Crawlers. *ACM Transactions on Database Systems*, 28(4):390–426, 2003. doi:10.1145/958942.958945.

[27] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1-7):161–172, 1998. doi:10.1016/S0169-7552(98)00108-1.

[28] Hyeong-Ah Choi and Bhagirath Narahari. Algorithms for Mapping and Partitioning Chain Structured Parallel Computations. In *International Conference on Parallel Processing*, pages 625–628, 1991.

[29] King-Wai Chow and Bede Liu. On Mapping Signal Processing Algorithms to a Heterogeneous Multiprocessor System. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 3 of *ICASSP '91*, pages 1585–1588, Washington, DC, USA, 1991. IEEE Computer Society. doi:10.1109/ICASSP.1991.150555.

[30] Chiasen Chung and Charles L. A. Clarke. Topic-Oriented Collaborative Crawling. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, pages 34–42, New York, NY, USA, 2002. ACM. doi:10.1145/584792.584802.

[31] CMU-LTI. The ClueWeb09 dataset. `http://lemurproject.org/clueweb09/`, 2009. Accessed: 2013-08-23.

[32] M. Coli and P. Palazzari. Real time pipelined system design through simulated annealing. *J. Syst. Archit.*, 42(6-7):465–475, December 1996. doi:10.1016/S1383-7621(96)00034-3.

[33] Jason Cong and M'Lissa Smith. A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design. In *Proceedings of the 30th Conference on Design Automation*, pages 755–760, 1993. doi:10.1109/DAC.1993.204048.

[34] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts, USA, 1989.

[35] Ernest Davis and Jeffrey M. Jaffe. Algorithms for Scheduling Tasks on Unrelated Processors. *Journal of the ACM*, 28:721–736, October 1981. doi:10.1145/322276.322284.

[36] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. doi:10.1145/2049662.2049663. `http://www.cise.ufl.edu/research/sparse/matrices`.

[37] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New Challanges in Dynamic Load Balancing. *Applied Numerical Mathematics*, 52(2–3):133–152, 2005. doi:10.1016/j.apnum.2004.08.028.

[38] NASA Advanced SuperComputing Division. NASA Advanced Supercomputing Division (NAS) Dataset Archive. `http://www.nas.nasa.gov/Research/Datasets/datasets.html`.

[39] Rubing Duan, Radu Prodan, and Thomas Fahringer. Performance and Cost Optimization for Multiple Large-scale Grid Workflow Applications. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 12:1–12:12, New York, NY, USA, 2007. ACM. doi:10.1145/1362622.1362639.

[40] José Exposto, Joaquim Macedo, António Pina, Albano Alves, and José Rufino. Geographical Partition for Distributed Web Crawling. In *Proceedings of the 2005 Workshop on Geographic Information Retrieval*, GIR '05, pages 55–60, New York, NY, USA, 2005. ACM. doi:10.1145/1096985.1096999.

[41] José Exposto, Joaquim Macedo, António Pina, Albano Alves, and José Rufino. Efficient Partitioning Strategies for Distributed Web Crawling. In Teresa Vazão, Mário M. Freire, and Ilyoung Chong, editors, *Information Networking. Towards Ubiquitous Networking and Services*, volume 5200 of *Lecture Notes in Computer Science*, pages 544–553. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-89524-4_54.

[42] Ivan De Falco, Renato Del Balio, Ernesto Tarantino, and Roberto Vaccaro. Improving Search by Incorporating Evolution Principles in Parallel Tabu Search. In *Proceedings of the First IEEE International Conference on*

Evolutionary Computation, volume 2 of *IEEE World Congress on Computational Intelligence*, pages 823–828, 1994. doi:10.1109/ICEC.1994.349949.

[43] David Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, November 1989. doi:10.1109/32.41334.

[44] Dennis Fetterly, Nick Craswell, and Vishwa Vinay. The Impact of Crawl Policy on Web Search Effectiveness. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 580–587, New York, NY, USA, 2009. ACM. doi:10.1145/1571941.1572041.

[45] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press. doi:10.1109/DAC.1982.1585498.

[46] Greg N. Frederickson. Optimal algorithms for partitioning trees and locating p-centers in trees. Technical Report CSD-TR-1029, Purdue University, 1990.

[47] Greg N. Frederickson. Optimal Algorithms for Tree Partitioning. In *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, SODA '91, pages 168–177, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[48] Richard F. Freund, Michael Gherrity, Stephen Ambrosius, Mark Campbell, Mike Halderman, Debra Hensgen, Elaine Keith, Taylor Kidd, Matt Kussow, John D. Lima, Francesca Mirabile, Lantz Moore, Brad Rust, and Howard Jay Siegel. Scheduling Resources in Multi-user, Heterogeneous, Computing Environments with SmartNet. In *Proceedings of the Seventh IEEE Heterogeneous Computing Workshop*, HCW '98, pages 184–199, Washington, DC, USA, 1998. IEEE Computer Society. doi:10.1109/HCW.1998.666558.

[49] Weizheng Gao, Hyun Chul Lee, and Yingbo Miao. Geographically Focused Collaborative Crawling. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 287–296, New York, NY, USA, 2006. ACM. doi:10.1145/1135777.1135822.

[50] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[51] Michael P. Garrity. Raytracing Irregular Volume Data. *ACM SIGGRAPH Computer Graphics*, 24(5):35–40, November 1990. doi:10.1145/99308.99316.

[52] M.K. Goldberg and M. Burstein. Heuristic improvement techniques for bisection of vlsi networks. In *Proceedings of the IEEE International Conference on Computer Design*, pages 122–125, 1983.

[53] Google. Google DNS Server. `https://developers.google.com/speed/public-dns`.

[54] Yijie Han, Bhagirath Narahari, and Hyeong-Ah Choi. Mapping a Chain Task to Chained Processors. *Information Processing Letters*, 44(3):141–148, 1992. doi:10.1016/0020-0190(92)90054-Y.

[55] Pierre Hansen and Keh-Wei Lih. Improved Algorithms for Partitioning Problems in Parallel, Pipelined, and Distributed Computing. *IEEE Transactions on Computers*, 41(6):769–771, 1992. doi:10.1109/12.144628.

[56] Michael Hardy. Pareto's law. *The Mathematical Intelligencer*, 32:38–43, 2010. doi:10.1007/s00283-010-9159-2.

[57] Bruce Hendrickson and Robert Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM. doi:10.1145/224170.224228.

[58] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. *World Wide Web*, 2(4):219–229, 1999. doi:10.1023/A:1019213109274.

[59] Ellis Horowitz and Sartaj Sahni. Exact and Approximate Algorithms for Scheduling Nonidentical Processors. *Journal of the ACM*, 23(2):317–327, April 1976. doi:10.1145/321941.321951.

[60] Ellis Horowitz and Sartaj Sahni. *Fundamentals of data structures*. Computer Software Engineering Series. Computer Science Press, 1983.

[61] Bradley Huffaker, Marina Fomenkov, Daniel J. Plummer, David Moore, and Kimberly C. Claffy. Distance Metrics in the Internet. In *IEEE International Telecommunications Symposium*, ITS 2002, pages 200–202, September 2002.

[62] Ching-Mao Hung and Pieter G. Buning. Simulation of Blunt-Fin-Induced Shock-Wave and Turbulent Boundary-Layer Interaction. *Journal of Fluid Mechanics*, 154:163–185, May 1985. doi:10.1017/S0022112085001471. `http://www.nas.nasa.gov/Research/Datasets/datasets.html`.

[63] Oscar H. Ibarra and Chul E. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM*, 24(2):280–289, April 1977. doi:10.1145/322003.322011.

[64] Mohammad Ashraf Iqbal. Approximate Algorithms for Partitioning and Assignment Problems. Technical Report 86-40, ICASE, 1986.

[65] Mohammad Ashraf Iqbal and Shahid H. Bokhari. Efficient Algorithms for a Class of Partitioning Problems. Technical Report 90-49, ICASE, 1990.

[66] Mohammad Ashraf Iqbal and Shahid H. Bokhari. Efficient Algorithms for a Class of Partitioning Problems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):170–175, February 1995. doi:10.1109/71.342129.

[67] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multi-level Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, March 1999. doi:10.1109/92.748202.

[68] Kamer Kaya and Cevdet Aykanat. Iterative-Improvement-Based Heuristics for Adaptive Scheduling of Tasks Sharing Files on Heterogeneous Master-Slave Environments. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):883–896, August 2006. doi:10.1109/TPDS.2006.105.

[69] Kamer Kaya, Bora Uçar, and Cevdet Aykanat. Heuristics for Scheduling File-Sharing Tasks on Heterogeneous Systems with Distributed Repositories. *Journal of Parallel and Distributed Computing*, 67(3):271–285, 2007. doi:10.1016/j.jpdc.2006.11.004.

[70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical J.*, 49:291–307, Feb. 1970.

[71] Vladimir Kolmogorov. Blossom V: A New Implementation of a Minimum Cost Perfect Matching Algorithm. *Mathematical Programming Computation*, 1(1):43–67, 2009. doi:10.1007/s12532-009-0002-8.

[72] Hüseyin Kutluca, Tahsin M. Kurç, and Cevdet Aykanat. Image-Space Decomposition Algorithms for Sort-First Parallel Volume Rendering of Unstructured Grids. *The Journal of Supercomputing*, 15(1):51–93, 2000. doi:10.1023/A:1008169609963.

[73] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. IRLbot: Scaling to 6 Billion Pages and Beyond. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 427–436, New York, NY, USA, 2008. ACM. doi:10.1145/1367497.1367556.

[74] Vitus J. Leung, Esther M. Arkin, Michael A. Bender, David Bunde, Jeanette Johnston, Alok Lal, Joseph S. B. Mitchell, Cynthia Phillips, and

Steven S. Seiden. Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '02, pages 296–304, Washington, DC, USA, 2002. IEEE Computer Society. doi:10.1109/CLUSTR.2002.1137758.

[75] Cong Liu and Sanjeev Baskiyar. A General Distributed Scalable Grid Scheduler for Independent Tasks. *Journal of Parallel and Distributed Computing*, 69, 2009. doi:10.1016/j.jpdc.2008.11.003.

[76] Max Otto Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9(70):209–219, Jun 1905.

[77] Ping Luo, Kevin Lü, and Zhongzhi Shi. A Revisit of Fast Greedy Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 67:695–714, 2007. doi:10.1016/j.jpdc.2007.03.003.

[78] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, 1999. doi:10.1006/jpdc.1999.1581.

[79] Fredrik Manne and Bjørn Olstad. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995. doi:10.1109/12.475128.

[80] MaxMind, Inc. GeoLite City. `http://www.maxmind.com/`, 2010.

[81] Serge Miguet and Jean-Marc Pierson. Heuristics for 1D Rectilinear Partitioning as a Low Cost and High Quality Answer to Dynamic Load Balancing. In Bob Hertzberger and Peter Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 550–564. Springer Berlin Heidelberg, 1997. doi:10.1007/BFb0031628.

[82] Gordon Mohr, Micheal Stack, Igor Ranitovic Dan Avery, and Michele Kimpton. Introduction to Heritrix: An Archival Quality Web Crawler. In

Proceedings of the 4th International Web Archiving Workshop, IWAW '04, September 2004.

[83] Marc Najork and Janet L. Wiener. Breadth-First Search Crawling Yields High-Quality Pages. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 114–118, New York, NY, USA, 2001. ACM. doi:10.1145/371920.371965.

[84] David M. Nicol. Rectilinear Partitioning of Irregular Data Parallel Computations. *Journal of Parallel and Distributed Computing*, 23(2):119–134, 1994. doi:10.1006/jpdc.1994.1126.

[85] David M. Nicol and David R. O'Hallaron. Parallel Algorithms for Mapping Pipelined and Parallel Computations. Technical Report 88-2, ICASE, 1988.

[86] David M. Nicol and David R. O'Hallaron. Improved Algorithms for Mapping Pipelined and Parallel Computations. *IEEE Transactions on Computers*, 40(3):295–306, 1991. doi:10.1109/12.76406.

[87] Christopher Olston and Marc Najork. Web Crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010. doi:10.1561/1500000017.

[88] Christopher Olston and Sandeep Pandey. Recrawl Scheduling Based on Information Longevity. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 437–446, New York, NY, USA, 2008. ACM. doi:10.1145/1367497.1367557.

[89] Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Multi-Heuristic Dynamic Task Allocation Using Genetic Algorithms in a Heterogeneous Distributed System. *Journal of Parallel and Distributed Computing*, 70:758–766, 2010. doi:10.1016/j.jpdc.2010.03.011.

[90] Saeed Parsa and Reza Entezari-Maleki. RASA-A New Grid Task Scheduling Algorithm. *International Journal of Digital Content Technology and its Applications*, 3(4):91–99, December 2009.

[91] John R. Pilkington and Scott B. Baden. Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, 1996. doi:10.1109/71.491582.

[92] Ali Pınar and Cevdet Aykanat. Sparse Matrix Decomposition with Optimal Load Balancing. In *Proceedings of the Fourth International Conference on High-Performance Computing*, HIPC '97, pages 224–229, Washington, DC, USA, 1997. IEEE Computer Society. doi:10.1109/HIPC.1997.634497.

[93] Ali Pınar and Cevdet Aykanat. Fast Optimal Load Balancing Algorithms for 1D Partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004. doi:10.1016/j.jpdc.2004.05.003.

[94] Frédéric Pinel, Bernabé Dorronsoro, and Pascal Bouvry. Solving Very Large Instances of the Scheduling of Independent Tasks Problem on the GPU. *Journal of Parallel and Distributed Computing*, 73(1):101–110, January 2013. doi:10.1016/j.jpdc.2012.02.018.

[95] Graham Ritchie and John Levine. A Hybrid Ant Algorithm for Scheduling Independent Jobs in Heterogeneous Computing Environments. In *PlanSIG 2004*, 2004.

[96] S. E. Rogers, D. Kwak, and U. K. Kaul. A Numerical Study of Three-dimensional Incompressible Flow Around Multiple Posts. In *AIAA 24th Aerospace Sciences Conference*, New York, NY, USA, 1986. American Institute of Aeronautics and Astronautics. doi:10.2514/6.1986-353. `http://www.nas.nasa.gov/Research/Datasets/datasets.html`.

[97] Prashanth C. SaiRanga and Sanjeev Baskiyar. A Low Complexity Algorithm for Dynamic Scheduling of Independent Tasks onto Heterogeneous Computing Systems. In *Proceedings of the 43rd Annual Southeast Regional Conference*, volume 1 of *ACM-SE 43*, pages 63–68, New York, NY, USA, 2005. ACM. doi:10.1145/1167350.1167380.

[98] D. G. Schweikert and B. W. Kernighan. A proper Model for the Partitioning of Electrical Circuits. In *Proceedings of the 9th Design Automation Workshop*, DAC '72, pages 57–62, New York, NY, USA, 1972. ACM. doi:10.1145/800153.804930.

[99] Hyunchul Shin and Chunghee Kim. A Simple Yet Effective Technique for Partitioning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(3):380–386, 1993. doi:10.1109/92.238449.

[100] Peter Shirley and Allan Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM SIGGRAPH Computer Graphics*, 24(5):63–70, November 1990. doi:10.1145/99308.99322.

[101] Sameer Shivle, Prasanna Sugavanam, Howard Jay Siegel, Anthony A. Maciejewski, Tarun Banka, Kiran Chindam, Steve Dussinger, Andrew Kutruff, Prashanth Penumarthy, Prakash Pichumani, Praveen Satyasekaran, David Sendek, Jay Smith, J. Sousa, Jayashree Sridharan, and Jose Velazco. Mapping Subtasks with Multiple Versions on an ad hoc Grid. *Parallel Computing*, 31(7):671–690, 2005. doi:10.1016/j.parco.2005.04.003.

[102] Vladislav Shkapenyuk and Torsten Suel. Design and Implementation of a High-Performance Distributed Web Crawler. In *Proceedings of the 18th International Conference on Data Engineering*, pages 357–368, Washington, DC, USA, 2002. IEEE Computer Society. doi:10.1109/ICDE.2002.994750.

[103] Howard Jay Siegel and Shoukat Ali. Techniques for Mapping Tasks to Machines in Heterogeneous Computing Systems. *Journal of Systems Architecture*, 46(8):627–639, 2000. doi:10.1016/S1383-7621(99)00033-8.

[104] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *Proceedings of the Eighth IEEE Heterogeneous Computing Workshop*, HCW '99, pages 3–14, 1999. doi:10.1109/HCW.1999.765092.

[105] Lee Wang, Howard Jay Siegel, Vwani R. Roychowdhury, and Anthony A. Maciejewski. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, November 1997. doi:10.1006/jpdc.1997.1392.

[106] J. L. Wolf, M. S. Squillante, P. S. Yu, J. Sethuraman, and L. Ozsen. Optimal Crawling Strategies for Web Search Engines. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 136–147, New York, NY, USA, 2002. ACM. doi:10.1145/511446.511465.

[107] Min-You Wu and Wei Shu. A High-Performance Mapping Algorithm for Heterogeneous Computing Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 74–79, Washington, DC, USA, April 2001. IEEE Computer Society. doi:10.1109/IPDPS.2001.925020.

[108] Fatos Xhafa, Enrique Alba, Bernabé Dorronsoro, and Bernat Duran. Efficient Batch Job Scheduling in Grids Using Cellular Memetic Algorithms. *Journal of Mathematical Modelling and Algorithms*, 7(2):217–236, 2008. doi:10.1007/s10852-008-9076-y.

[109] Demetrios Zeinalipour-Yazti and Marios D. Dikaiakos. Design and Implementation of a Distributed Crawler and Filtering Processor. In Alon Halevy and Avigdor Gal, editors, *Next Generation Information Technologies and Systems*, volume 2382 of *Lecture Notes in Computer Science*, pages 58–74. Springer Berlin Heidelberg, 2002. doi:10.1007/3-540-45431-4_6.

Appendix A

Detailed Analysis

A.1 Average Case Analysis of NICOL+

Below, we will provide average case analysis of `NICOL+`, this analysis prove that, the worst-case runtime complexity of $O(N+K^2 \log N \log(1+w_{\max}/(Ke_{\min}w_{\min})))$ reduces to $O(N + K \log K \log N \log(1 + w_{\max}/(e_{\min}w_{\min} \log K)))$ in the average case, which is an asymptotical improvement of $O(K/\log K)$. This analysis also valid for homogenous `NICOL+` algorithm, with $e_{\min} = 1$.

Lemma A.1.1 *On average, less than* $1 + \ln K$ *of the processors update the* `LB` *value.*

Proof: Assume the bottleneck values produced by Nicol's algorithm is given by the sequence $\langle B_1, B_2, \ldots, B_K \rangle$. A B_p bottleneck value of processor $\mathcal{P}_p$ updates `LB` iff $B_p > B_i$ for all $1 \leq i < p$.

Assuming the order of B_p values are randomly distributed, the problem is simplified to the expected number of up-records in a random permutation, EL_K

which can be effectively computed as:

$$\begin{aligned} EL_K &= \Pr(B_1 \text{ is a record}) + \Pr(B_2 \text{ is a record}) \\ &\quad + \ldots + \Pr(B_K \text{ is a record}) \\ &= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{K} \\ &< 1 + \ln K \end{aligned}$$

So, on average, less than $1 + \ln K$ of the processors update the LB value. ■

Lemma A.1.2 *On average, less than* $1 + \ln K$ *of the processors update the UB value.*

Proof: Proof is similar to that of Lemma A.1.1. ■

Corollary A.1.1 *On average, during an execution of* `NICOL+`*, less than* $2+2\ln K$ *of the processors requires* `PROBE` *calls.*

Lemma A.1.3 *On average, for K processors,* `NICOL+` *require no more than* $O(\log K \log(1 + w_{\max}/(w_{\min} e_{\min} \log K)))$ `PROBE` *calls.*

Proof: In average case, less than $2 + \ln K$ of those processors call `PROBE` functions.

The rest of the proof is similar to Lemma 3.3.4. Consider the first step of the algorithm, where we search for the smallest separator index that makes the first processor the bottleneck processor. We can restrict this search in a range that covers only those indices for which the weight of the first chain will be in the $[LB, UB]$ interval. If there are n_1 tasks in this range, `NICOL+` will require $\lg n_1$ probes. This means that the $[LB, UB]$ interval is narrowed by at least $(n_1 - 1) w_{\min}$ after the first step.

Let k_p be the number of probes by the pth processor. Since k_p probes narrow the $[LB, UB]$ interval by $\left(2^{k_p} - 1\right) w_{\min}$, and less than $2 + \ln K$ processors call

probes, we have

$$\left(\left(2^{k_1}-1\right)+\left(2^{k_2}-1\right)+\ldots+\left(2^{k_{K'}}-1\right)\right) w_{min} \leq UB - LB, \tag{A.1}$$

where $K' = \lceil 2 + \ln K \rceil$. Thus,

$$2^{k_1}+2^{k_2}+\ldots+2^{k_{K'}} \leq \frac{UB-LB}{w_{\min}} + K'. \tag{A.2}$$

The corresponding total number of probes is

$$\sum_{p=1}^{K'} k_p, \tag{A.3}$$

which reaches its maximum when

$$\sum_{p=1}^{K'} 2^{k_p} \tag{A.4}$$

is maximum and

$$k_1 = k_2 = \ldots = k_{K'} = k \tag{A.5}$$

for some k. In that case,

$$K' 2^k \leq \frac{UB-LB}{w_{\min}} + K' \tag{A.6}$$

and thus

$$k \leq \lg\left(1+\frac{UB-LB}{w_{\min}K'}\right). \tag{A.7}$$

So, the total number of probes performed by `NICOL+` is upper bounded by:

$$\begin{aligned}
\sum_{p=1}^{K'} k_p &\leq K'k && \text{(A.8)}\\
&\leq K' \lg\left(1+\frac{UB-LB}{w_{\min}K'}\right) && \text{(A.9)}\\
&= \lceil 2+\ln K\rceil \lg\left(1+\frac{\frac{w_{\max}}{e_{\min}}-\frac{w_{\max}}{Ke_{\min}}}{w_{\min}\lceil 2+\ln K\rceil}\right) && \text{(A.10)}\\
&= O\left(\log K \log\left(1+\frac{w_{\max}}{e_{\min}w_{\min}\log K}\right)\right). && \text{(A.11)}
\end{aligned}$$

The number of probes performed by $\lceil 2 + \ln K \rceil$ processors is bounded by $O(\log K \log(1 + w_{\max}/(e_{\min} w_{\min} \log K)))$. ■

With the $O(K \log N)$ cost of a `PROBE`, the expected runtime complexity of `NICOL+` becomes $O(N + K \log N \log K \log(1 + w_{\max}/(e_{\min} w_{\min} \log K)))$. If $e_{\min} \log K = \Omega(w_{\max}/w_{\min})$, then the complexity reduces to $O(N+K \log N \log K)$.

Appendix B

Code

The algorithms proposed in Chapter 3 are implemented in Java language and made publicly available at `http://www.cs.bilkent.edu.tr/~tabak/hetccp/`. Other algorithms are available upon request.

Appendix C

The Process of Generation of ETC Matrices for ClueWeb-09 datasets

This chapter describes the processes used to generate the task-assignment Expected-Time-To-Complete (ETC) matrices for large ClueWeb-09 datasets [31]. The output of this process is used in various chapters of this thesis. The mechanism can be thought as several layers of MapReduce procedures, each layer summarizing a kind of information on the large dataset.

C.1 Outline

The outline for this generation is as follows:

- Obtain the Crawl Data
- Parse Crawl Data to obtain multi-part hosts and links information (WarcProcessor1)
- Merge multi-part hosts information into a single hosts file

C.2 Obtain the Crawl Data

A copy of the ClueWeb-09 data can be obtained from Carnegie Mellon University. The license for Computer Science Department of Bilkent University is obtained by Prof. Özgür Ulusoy. The data is in the form of four 1.5TB hard disks, and each disk contains many folders, subfolders and files in .warc.gz format of the crawl data.

Note that, the hard disks contains compressed version of the data. The compressed version is around 5TB. Uncompressed version of this data is more than 30TB!

C.3 WarcProcessor1

First mount the obtained crawl data disk to some place. Be careful, there are no backup copies of these disks. Mount these disks on a linux machine, with read-only access. The format of the disks are ext4.

Then execute WarcProcessor1 executable. This process gets warc root directory and outputs the hosts and links information in a separate file for each of the datasets. We also maintain a status directory for this process. This status directory provides some kind of job distribution service for multi-process and multi-machine execution of this phase. At the end of this phase we obtain hosts and links information distributed in several files and directories.

From the dataset, we parse only "response" type warc information. Hosts information include:

- Source Host Name
- 1 (To represent one page)
- Response Length

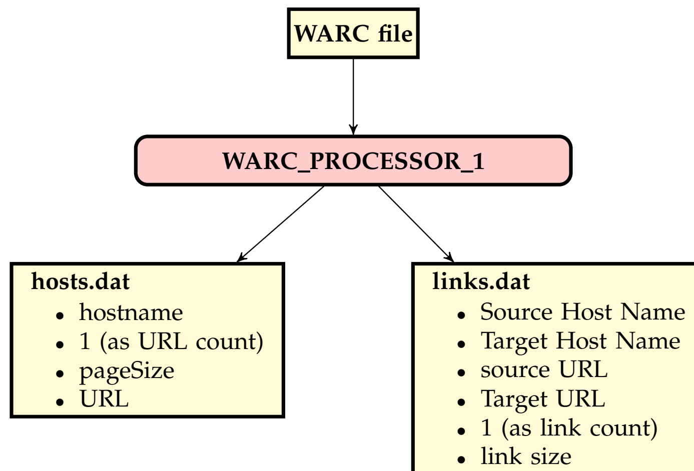


Figure C.1: WarcProcessor1 data flow

- Source URL

For the link extraction, we utilize `ExtractorHTML` processor class from `org.archive.crawler.extractor` package of heritrix [82] project. To perform a safe link extraction, we ignore the links with:

- unrecognized source URL
- links containing '\n' character.
- links containing '\r' character.
- links containing '\t' character.
- links containing space character.

Links information include:

- Source Host Name
- Target Host

- Source URL
- Target URL
- 1 (To represent one link)
- Target URL Length

The size of the output of this step is around 1TB.

The data flow diagram of a single WARC file for WarcProcessor1 is presented in Fig. C.1.

C.4 WarcProcessor5

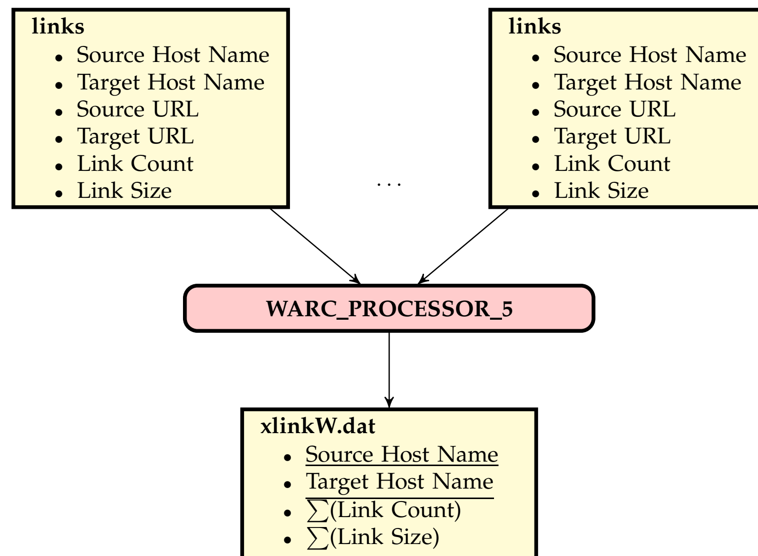


Figure C.2: WarcProcessor5 data flow diagram.

This process merges the links information gathered in Appendix C.3 into a single link file. Link size is recalculated in the process. Link size is defined as the sum of the lengths of source and target URLs.

WarcProcessor5B also defines another method of merging link information.

C.5 WarcProcessor5B

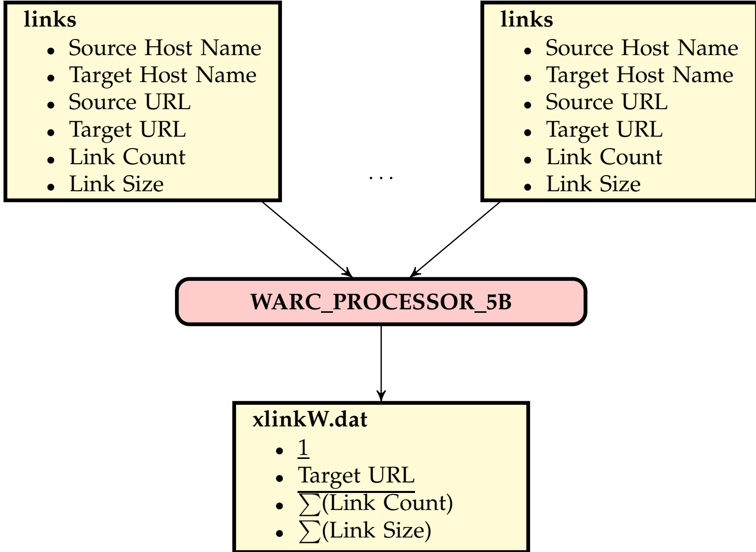


Figure C.3: WarcProcessor5B data flow diagram.

This process is very similar to the WarcProcessor5. The main difference is the output key, which contains the target URL, instead of source host name and target host name pair. The result is a bigger output file, which has the incoming link information for all of the known URLs. Obvious advantage of using that method over WarcProcessor5 is having an accurate number of incoming link counts for all URLs on the crawl dataset.

As in WarcProcessor5, Link size is recalculated within the process. Link size is defined as the sum of the lengths of source and target URLs.

C.6 WarcProcessor2

In this process, we merge the hosts information splitted on several files into a single hosts file. At the end of this process, we obtain basic information of hosts. The output file is sorted by host names, see Fig. C.4.

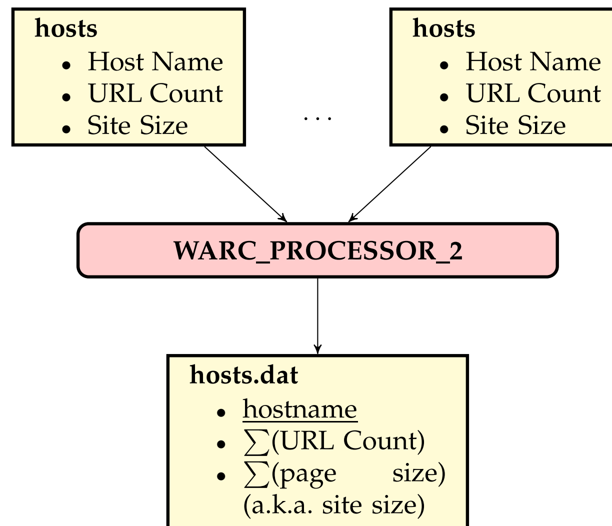


Figure C.4: WarcProcessor2 data flow diagram. WarcProcessor2 merges host files into a single sorted hosts file

Hosts information include:

- Source Host Name
- URL Count
- Site Size: total response length

Since the dataset is quite big that cannot fit into memory, we utilize some disk merging techniques to merge the hosts data. To outcome memory limitation, we have implemented an external K-way mergesort [60]. We read files one-by-one, and output the results to a temporary file when we reach a certain number of hosts information. Each of the temporary files contains a predefined number of hosts, in sorted order. These files are then merged with groups (of 10) and we obtain a larger sorted file. If we encounter the same host information on different files, we merge the contents so that at the end we reach only one host information for each host.

C.7 WarcProcessor1B

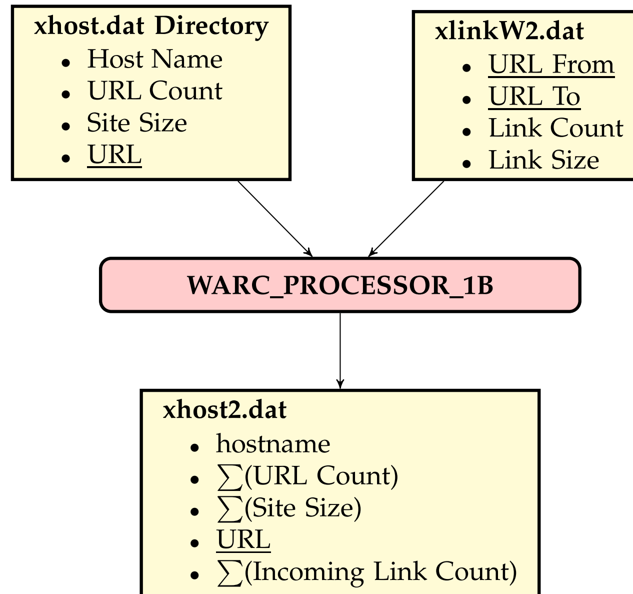


Figure C.5: WarcProcessor1B data flow diagram.

Fig. C.5 describes the data flow of WarcProcessor1b. WarcProcessor1b has two inputs and one output. First input is the directory which contains xhost.dat files produced by Appendix C.3. The second input is xlinkW2.dat produced by Appendix C.5. The output is xhost2.dat.

The process is similar to that of WarcProcessor2, the main difference is the input file, which is produced by WarcProcessor5B instead of WarcProcessor5.

C.8 WarcProcessor1c

Fig. C.6 describes the process of obtaining a file called xhost3.dat. The processor takes xhost2.dat produced by WarcProcessor1B which contains the individual URLs of pages, along with host names, URL counts (probably 1, but may vary), page sizes, and incoming link counts.

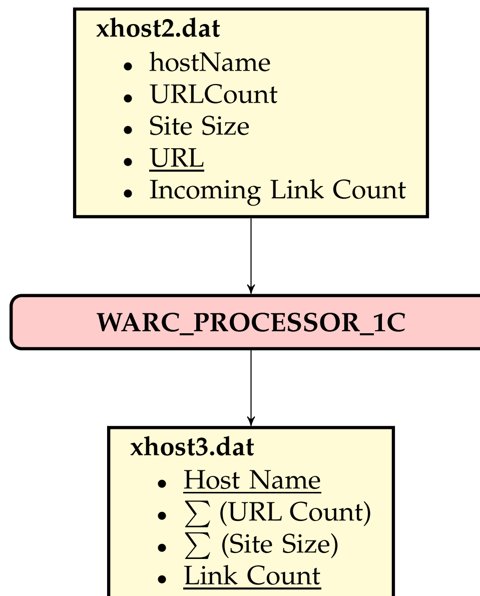


Figure C.6: WarcProcessor1C data flow diagram.

The output is a summary of these pages, grouped by host name, and link counts. URLs of a host name with no incoming edges are accumulated into a single row, URLs of the same host name with 1 incoming edge are accumulated into another single row, etc.

C.9 WarcProcessor3

Fig. C.7 describes the process of WarcProcessor3. The processor takes base-host.dat which contains the hostName information and outputs the host name - IP map.

Internally, WarcProcessor3 has a number of IP Resolvers which are executed on a layered machanism. If IP can be resolved on one layer, it does not fall through the next layer. However, if IP cannot be resolved on the current layer, the host name falls through the next layer with the hope to resolve it on the next layer. The figure describing the process is presented on Fig. C.8.

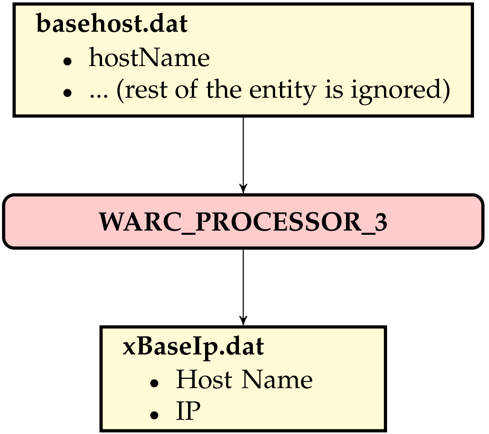


Figure C.7: WarcProcessor3 data flow diagram.

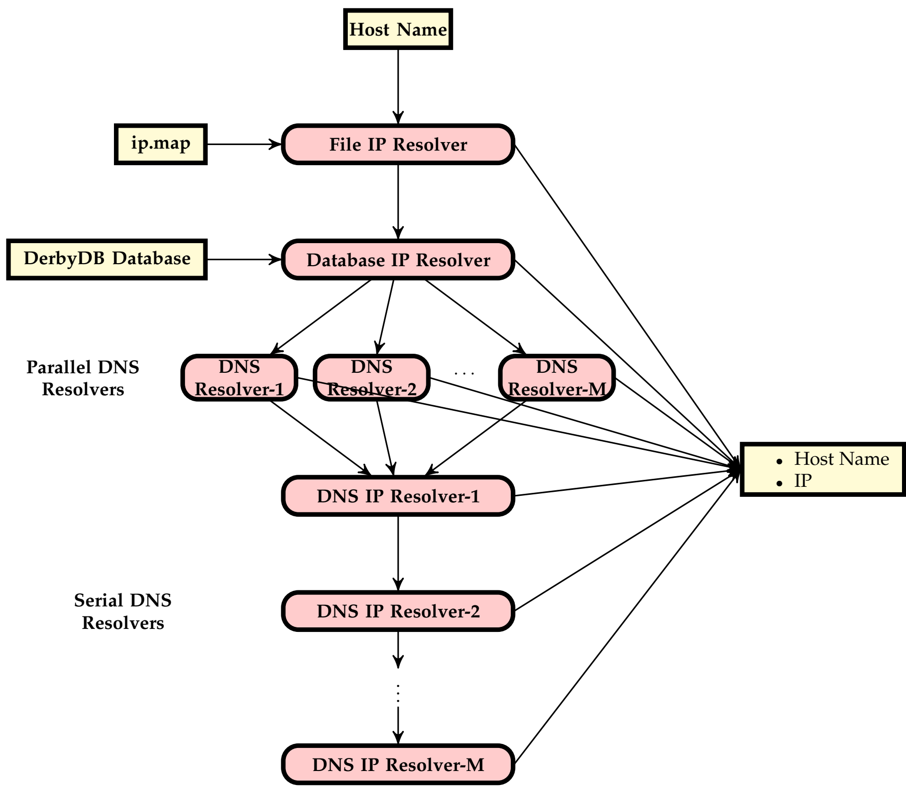


Figure C.8: WarcProcessor3 internal data flow diagram.

The first layer is a file IP map. This map is a file based map that can fit into memory. This layer is feeded by ip.map file that is small enough to be represented by a `HashMap` in the local memory. If this layer cannot find the IP of the host name, the result is retried on next layer.

The next layer is a database map. The database is implemented by the database Apache DerbyDB [4].

The next layer is parallel DNS IP resolvers. There may be more than one DNS servers. Free ones (e.g. Google DNS [53]) are slow but does not complain for the load. Local DNS servers are generally expected to be faster, but be careful: If your DNS server is not tuned for bulk queries, you may slow down local DNS server and thus the whole internet access of your domain. Hence, we have many DNS servers which are executing in parallel. In this layer, a host-to-ip task is executed on only one DNS resolver. If that resolver fails to find a mapping, the host name falls through the next layer.

Following the parallel DNS IP resolvers, there are a series of layers for serial DNS IP resolvers. The host names that cannot be found on the previous layers are retried on each of the DNS IP resolvers, one by one. If the resolver cannot find on its server, the resolver delivers the host name to the next DNS IP resolver.

If none of the DNS IP resolvers can find the IP of the host name, we assume the host name as invalid.

Note that, all of those resolvers are implemented as parallel processing queues, to achieve highest throughput.

C.10 Ip2Geo

This process find the geographical locations of each host. The process utilizes the MaxMind IP-to-geo database [80]. This database claims to be accurate more than 99% at country level. The output of the process is a hosts file with host names, along with the IP of the host and the host country of the IP.

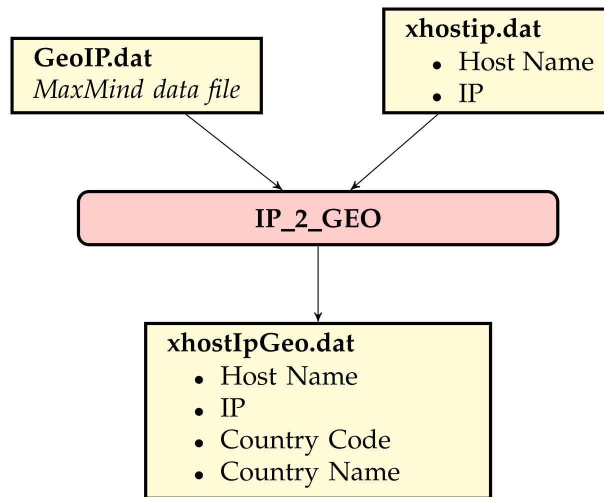


Figure C.9: Ip2Geo data flow diagram.

C.11 WarcProcessor6

This process produces processor.dat. The process finds the host countries of each server and sorts the countries with the amount of data they serve. The process selects the top K countries and assigns a data center with 10MB/sec bandwith for each country.

C.12 WarcProcessor7

This process produces assign-K.dat, which is the main input file format expected in many independent task assignment algorithms. countries.dat is a manually-typesetted country geographic-location lookup-file. xhostIpGeo.dat is the output of Appendix C.10. processor.dat is the output of Appendix C.11. The expected-time-to-crawl values are calculated using the formulas provided in Chapter 5, while WarcProcessor7 ignores the refresh frequency and assumes the frequencies as 1. The output of this process is utilized in Chapter 4.

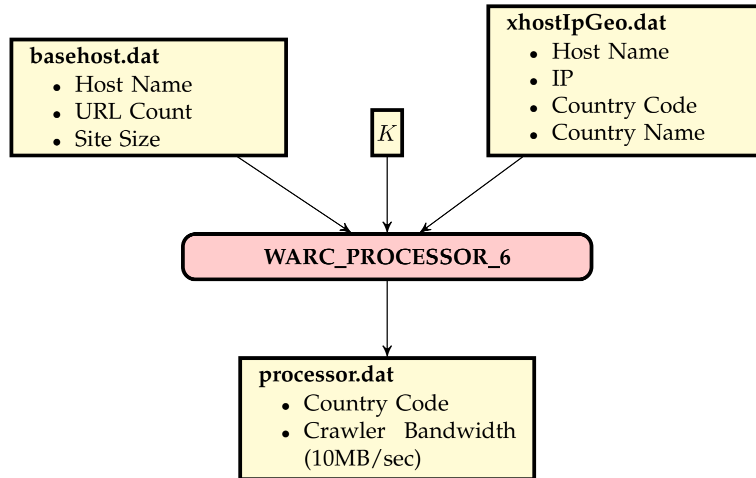


Figure C.10: WarcProcessor6 data flow diagram.

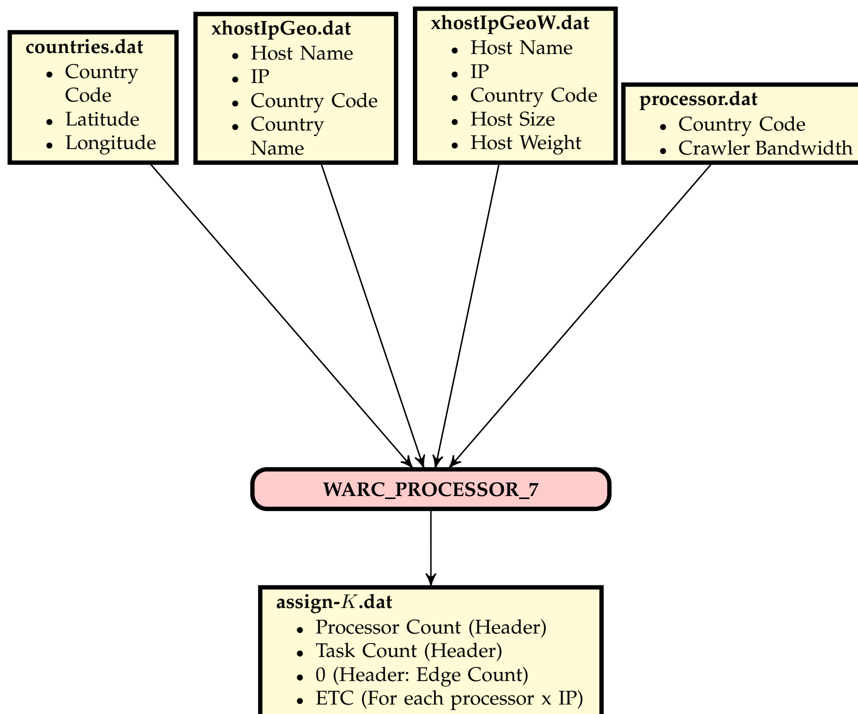


Figure C.11: WarcProcessor7 data flow diagram.

C.13 WarcProcessor7B

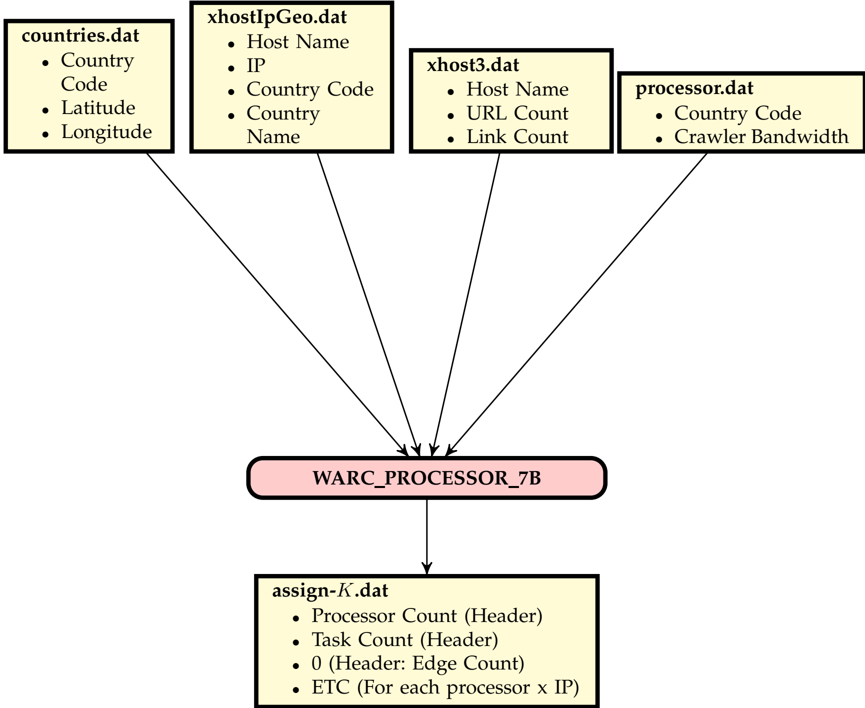


Figure C.12: WarcProcessor7B data flow diagram.

As in WarcProcessor7, this process produces assign-K.dat, which is the main input file format expected in many independent task assignment algorithms. countries.dat is a manually-typesetted country geographic-location lookup-file. xhostIpGeo.dat is the output of Appendix C.10. processor.dat is the output of Appendix C.11. Unlike WarcProcessor7, the expected-time-to-crawl values contain the refresh frequencies. The output of this process is utilized in Chapter 5.