A BIPARTITE GRAPH MODEL FOR PLACEMENT, SCHEDULING AND REPLICATION IN DATA GRIDS

A THESIS

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

MASTER OF SCIENCE

By

Burcu Dal

September, 2012

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

Prof. Dr. Cevdet Aykanat(Advisor)

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

Prof. Dr. Özgür Ulusoy

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

Asst. Prof. Dr. Sinan Gezici

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

A BIPARTITE GRAPH MODEL FOR PLACEMENT, SCHEDULING AND REPLICATION IN DATA GRIDS

Burcu Dal
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2012

Data grids provide geographically distributed resources for applications that generate and utilize large data sets. However, there are some issues that hinder to ensure fast access to data and low turnaround time for the jobs in data grids. To address these issues, several data replication and job scheduling strategies have been introduced to offer high data availability, low bandwidth consumption, and reduced turnaround time for grid systems. Multiple copies of existing data are maintained at different locations via data replication. Data replication strategies are broadly categorized as static and dynamic. In static replication strategies, replication is performed during the system design, and replica decisions are generally based on a cost model that includes data access costs, bandwidth characteristics and storage constraints of the grid system. In dynamic replication strategies, the replication operation is managed at runtime so that the system adapts to the changes in user request patterns dynamically. Job scheduling strategies fall under two main categories: online mode and batch mode. The online mode scheduler assigns tasks to sites as soon as they arrive. In the batch mode, the complete set of jobs are taken into account and scheduled at the same time by using all the grid information.

In this thesis, we propose a bipartite graph model for tasks and files in the grid system, and then we partition this graph to obtain a data placement and job scheduling strategy. The obtained parts are further refined in order to be assigned to grid sites by using a KL-based heuristic that takes the bandwidth and hop information between sites into account. Replication is achieved by replicating a certain amount of most accessed files chosen prior to the partitioning process. Experimental results indicate that the increase in the partitioning quality reflects positively on the mapping quality. Morever, it is observed that the communication cost is notably decreased when the data replication is applied. Hence, our

results show that by replicating a small amount of data files and placing files onto sites using bipartite graph model, we can obtain performance improvement for scheduling jobs compared to no replication.

Keywords: Data Grids, Bipartite Graph, Data Placement, Job Scheduling, Data Replication.

ÖZET

VERİ GRİDLERİNDE YERLEŞTİRME, ÇİZELGELEME VE ÇOKLAMA İÇİN İKİ-KISIMLI ÇİZGE MODELİ

Burcu Dal
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2012

Veri gridleri, büyük veri setleri üreten ve kullanan uygulamalar için coğrafi olarak dağıtılmış kaynaklar sağlar. Halbuki, veri gridlerinde veriye hızlı erişim ve işler için düşük yanıt süresi temin etme durumları, çeşitli sebeplerden dolayı engellenmektedir. Bu sorunları ele almak için, yüksek veri elverişliliği, düşük bant genişliği tüketimi ve indirgenmiş yanıt süresi sağlayan değişik veri çoklama ve iş çizelgeleme stratejileri sunulmuştur. Veri çoklama sayesinde, veri farklı konumlarda çok kopyalı şekilde muhafaza edilmektedir. Ayrıca, grid üzerinde etkili bir şekilde iş çizelgeleme yaparak, sistem verimliliğinin arttırılması amaçlanmıştır. Çoklama stratejileri genelde statik ve dinamik olarak sınıflandırılır. Statik çoklama stratejilerinde, çoklama kararları çoğunlukla grid sistemindeki veri erişim maliyetlerini, bant genişliği özelliklerini ve saklama kısıtlarını kapsayan bir maliyet modeline dayanarak verilir ve çoklama işlemi sistemin tasarlanması sırasında yapılmaktadır. Dinamik çoklama stratejilerinde çoklama işlemi, kullanıcı isteği deseninindeki değişiklikleri sisteme uyarlamak için çalışma zamanında yapılmaktadır. İş çizelgeleme stratejileri, çevrimiçi mod ve toplu mod olmak üzere iki genel kategorinin içinde yer alırlar. Çevrimiçi mod çizelgeleyicisi, bir işi ulaşır ulaşmaz bir makineye atar. Toplu mod yönteminde, bütün grid bilgisini kullanarak, bütün işler aynı anda ele alınır ve çizelgelenir.

Biz bu çalışmada, grid sistemindeki işleri ve verileri temsil eden bir "iki kısımlı çizge" modeli önermekteyiz. Veri yerleştirme ve iş çizelgeleme stratejisi elde etmek için bu çizgeyi bölüntülüyoruz. Elde edilen bölüntüler, yerleşkeler arasındaki bant genişliğini ve hoplama bilgisini hesaba katan KL-tabanlı buluşsal bir çizge bölüntüleme yöntemi kullanarak, grid yerleşkelerine atama yapmak için yeniden iyileştirilmektedir. Çoklama, bölüntüleme sürecinden önce seçilen en çok erişilen dosyaların belli bir miktarını kopyalarak gerçekleştirilir. Deneysel sonuçlar göstermektedir ki, bölüntüleme kalitesindeki artış, atama kalitesine

olumlu şekilde yansımaktadır. Buna ek olarak, veri çoklama uygulandığında iletişim maliyetinin dikkate değer ölçüde düştüğü gözlemlenmiştir.

Anahtar sözcükler: Veri Gridleri, İki-Kısımlı Çizge, Veri Yerleştirme, İş Çizelgeleme, Veri Çoklama.

Acknowledgement

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

I am thankful to Prof. Dr. Özgür Ulusoy and Asst. Prof. Dr. Sinan Gezici for kindly accepting to be in the committee and also for giving their precious time to read and review this thesis.

I am specially thankful to Oğuz Selvitopi for sharing his ideas, suggestions and valuable experiences with me throughout the year.

Comments given by Ata Türk and Enver Kayaaslan have been great help in my thesis study.

I would like to thank all of my friends for the enjoyable times during my master study. I wish to thank my best friend Esra Aktaş for her valuable friendship. Also, I would like to thank Murat Açar for his endless support and patience in writing the thesis and for caring and entertainment he provided.

Finally, most of my gratitude goes to my dearest family (Mehmet, Feriha, Elif Dal, Fatma Türkoğlu and Cemal Yekbaşlı). Their profound love, tremendous support and motivation led me to where I am today. To them, I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data intensive scientific applications in the fields of high-energy physics, bioinformatics, climate modeling and other related disciplines [1–3] have appreciably increased in importance. These applications are aimed at resolving central issues with which mankind have confronted, and they set up these problems on a sound basis of scientific research. One of the major considerations about data intensive applications is that they consume and produce several of data which is highly geographically dispersed in large number of files or objects. In general, a large amount of loosely coupled jobs associated with these applications are dependent on large-scale distributed data sets.

Technological advances in computational, storage and network units enable both practitioners and researchers to widen the sophistication and scope of data-intensive applications [4]. *Data grid* is an enabling technology for data intensive applications, and it consists of a large number of distributed computation and storage resources. Grid infrastructure manages large scale data files and provides computational resources across widely distributed communities. While handling a *data grid environment*, we certainly deal with a huge amount of metrics and constraints due to getting possession of potentially independent sources of jobs and a large number of storage, computation, and network resources. We are in need of effective scheduling and replication mechanisms for such environments that eventually turn out to be laborious tasks [5].

1.1 Scope of the Work

In our grid system, each site conventionally contains a storage and a computation unit, but there can be other grid systems where sites can contain either storage or computation units. A *data grid* can be represented as an *undirected graph* where vertices represent the sites and edges represent the connection between those sites. *Grid scheduler* places the data and schedules the incoming jobs to the sites in the system. In a *data grid*, the duty of the *grid scheduler* is more challenging compared to a typical scheduler since the data access characteristics as well as the computational characteristics of the execution site collectively determine the job execution efficiency. Therefore, both data and computational requirements of a job have effects on the scheduling decision for that job. As summarized in [6], *grid scheduler* is responsible for the following steps that should be performed seamlessly:

- Exploration of Resources is the task of revealing both the input data locations that are associated with a job and the storage resources for throughput.
- System Selection totally depends on the scheduling decision in which data access time for both input and output locations needs to be considered delicately. Also, efficient replication mechanisms can make a contribution in this task by feasibly replicating data.
- Job Execution is rather based on the fluctuations in network performance. Some snap decisions on the locations of data access and scheduling can be made, and we may have to face altered circumstances of scheduling and replication of data instantaneously.

Having stated the principal components of the architecture, Figure 1.1 depicts the overall architecture of *data grid*.

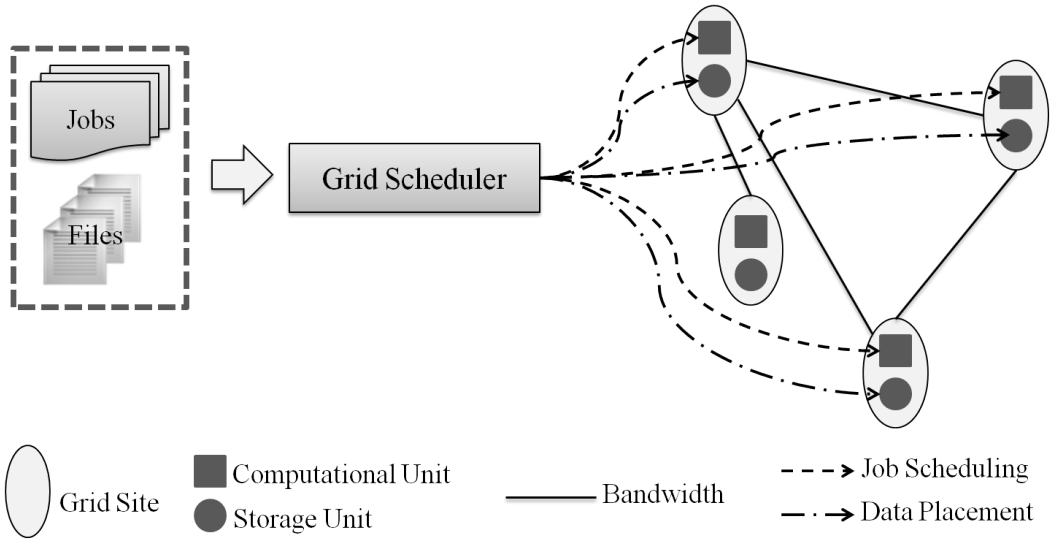

Figure 1.1: System architecture of the data grid

1.2 Motivation and Problem Statement

In the grid environment, it is difficult to ensure fast access to data and low turnaround time for jobs. To enhance the performance of a data grid, job scheduling and replication strategies have been introduced. *Job scheduling* is responsible for assigning jobs onto sites. *Replication* maintains multiple copies of existing data at different locations in the grid system to improve system performance and availability. Advantages of efficient job scheduling and data replication can be stated as follows:

- Reducing the bandwidth consumption
- Improving the performance of data access
- Reducing the access latency
- Minimizing the overall job execution time

The objective in response to the mentioned advantages is to exploit the synergies between data replication and job scheduling to achieve better system performance as we have the following earnings associated with these approaches:

- ✓ A good scheduling strategy will allow shortest access to the required data; thereby, the data access time will be decreased.
- ✓ A good replication strategy will offer faster access to files required by grid jobs, and job completion time will also be minimized.

The combination of these two strategies to increase the grid system performance is the major problem as the optimization of both data replication and job scheduling will be challenging. The questions that need to be addressed in this context are:

- ◇ How to formulate a problem that incorporates both objective functions in the same framework?
- ◇ How to address the issue of finding a good solution for both objectives?

Having considered the stated problems, our methodology consists of two major phases. Firstly, we assign files and jobs to the sites given access logs. Under this phase, we have two important steps. One step is to partition task-file graph into a given number of parts, and the second is to map these parts to the sites. Finally, we replicate selected files through all sites. These phases and related steps are given as follows:

1. Data Placement & Job Scheduling
   (a) Partitioning Task File Graph
   (b) Mapping of Parts to Sites
2. Data Replication

1.3 Thesis Outline

The thesis is organized as follows:

- ◇ Chapter 2 provides a background of *data intensive applications* and *data grids* along with the methods used.
- ◇ Chapter 3 presents the results of our literature survey on *data replication* and *job scheduling* in data grids with their integrated utilization.
- ◇ Chapter 4 focuses on a *bipartite graph model* for tasks and files, and a practicable *graph partitioning approach*.
- ◇ Chapter 5 is based on solving problem of "*mapping obtained parts to sites*" for which we have followed a *KL-based heuristic*.
- ◇ Chapter 6 shows our experimental results in terms of both partitioning and improvement based on the process of *data grid environment* and dataset generation.
- ◇ Chapter 7 concludes the study with our experiences throughout the work done, some lessons-learned and future work.

Chapter 2

Background

2.1 Data Intensive Applications

Data-intensive applications are I/O bound applications that require efficient manipulation of terabytes of data aggregated across hundreds of files. They involve the geographically dispersed extraction of complex scientific information from very large collections of measured or computed data. Therefore, the transfer of information in wide area and distributed computing environments is an important requirement that needs to be handled efficiently. Examples of such applications include experimental analysis, simulations and comparisons of outputs in scientific disciplines, such as high-energy physics, climate modeling, earthquake engineering, and astronomy [7].

CERN High-Energy Physics Experiments

As an example of data intensive applications, characteristics of high-energy physics experiments are analyzed below [7]:

- ◇ High-energy physics experiments produce several Petabytes of data per year over a life time of 15 to 20 years and then, they operate on this data.
- ◇ Data are written by the experiment, stored at very high data rates and

are generally not changed any more afterwards. Thus, one characteristic of such data is that most of it is read-only.

- ◇ The data generated by physics experiments is of two types:
  - Experimental data represents the information collected by the experiment. There is a single creator of this data, and once created, it is not modified. However, data may be collected incrementally over a period of weeks.
  - Metadata captures information about the experiment and the results of analysis. Multiple individuals may create metadata. The volume of metadata is typically smaller than that of experimental data.

  Access patterns vary for experimental data files and metadata.
- ◇ File sizes and numbers of files are usually determined by the type of software used to store experimental data and metadata. Current file sizes range from 2 to 10 gigabytes in size, while metadata files are around 2 gigabytes.

In Table 2.1, the characteristics of high-energy physics applications are summarized.

Rate of data generation (starting 2005)	Several petabytes per year
Typical experimental database file size	2-10 gigabytes
Typical metadata database file size	2 gigabytes
Period of updates to experimental data	Several weeks
Period of updates to metadata	Indefinite
Type of storage system	Object-oriented database
Number of data consumers	Hundreds to thousands

Table 2.1: Characteristics of high-energy physics applications

2.2 Data Grids

The data intensive applications involve operations on the geographically distributed, large data collections. A lot of researchers from different places in

the world need to access and analyze the results of these applications. Therefore, transfer of the information in distributed environments is an important requirement to be handled efficiently. The literature offers numerous point solutions that address these issues. However, no integrating architecture exists that allows us to identify requirements and components common to different systems and hence apply different technologies in a coordinated fashion to a range of data-intensive petabyte-scale application domains [7, 8].

Motivated by these considerations, grid computing has become an important and interesting new field of research since it offers a large variety of applications. Informally, a grid is a parallel and distributed system that enables dynamic sharing, selection, and aggregation of geographically distributed, autonomous resources, depending on their availability, capability, performance, cost, and user quality-of-service requirements. So, the main aim of a grid is to connect geographically distributed resources into one large system that enables users to have transparent access to data and computing resources across the grid. These resources are usually much bigger and powerful than the resources available at the users' local sites. Grids can be classified into computational grids and data grids. The main task of a computational grid is to manage computing resources and computationally intensive tasks. The main task of a data grid is to manage huge amounts of data and data intensive tasks [7–9].

Data grid infrastructure meets the needs of data intensive applications by connecting a collection of hundreds of geographically distributed computers and storage resources located in different parts of the world to facilitate sharing of data and resources. The size of data that needs to be accessed on a typical data grid is in the order of Terabytes [9].

This data grid structure is also suitable for CERN experiments since they are collaborations of over a thousand physicists from many different universities and institutes. Therefore, the experiment's data are not only stored locally at CERN but there is also an intention to store parts of the data at world-wide distributed sites in so-called Regional Centers and also in some institutes and universities [7].

A data grid can be supported by many different architectures:

◇ The **multi-tier architecture** is a tree like structure in which the nodes are arranged in a tree like hierarchy. For example, the data grid of the GriPhyN project [10] in which tier 0 is the main data source (CERN), tier 1 contains the national centers, tier 2 the regional centers, tier 3 the work groups and finally at tier 4 are the desktops. This is a form of client-server architecture and is easier to implement because of its simplicity. The problem with this type of architecture is the strict rules of a tree structure; there is only one path available from a leaf to the root. Child nodes can communicate only to their direct parent and cannot communicate with any other node. This type of model is efficient only for the grids which are designed from scratch. Figure 2.1 illustrates a multi-tier data grid architecture.

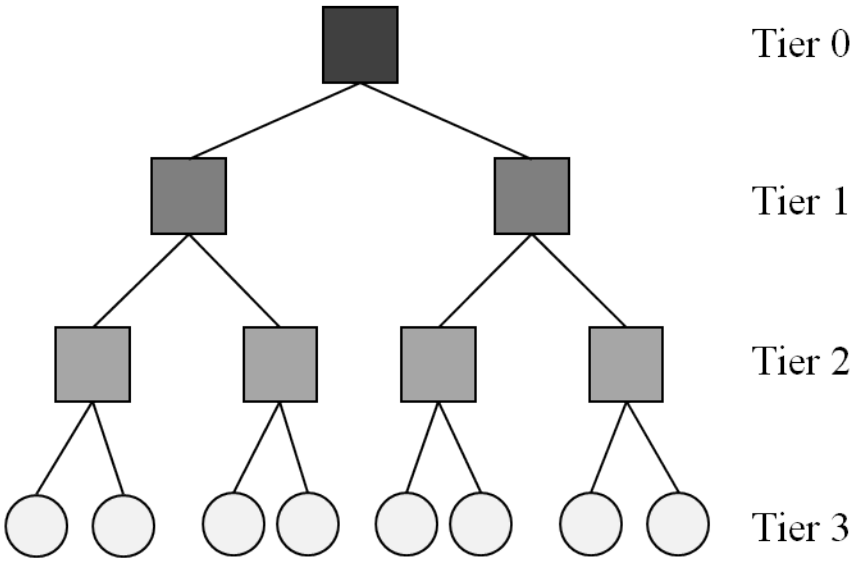

Figure 2.1: Multi-tier architecture

◇ The **sibling tree architecture** is a modification of this hierarchal model in which the sibling nodes are also connected. This improves some of the limitations of the hierarchical grid. Figure 2.2 illustrates a sibling tree data grid architecture.

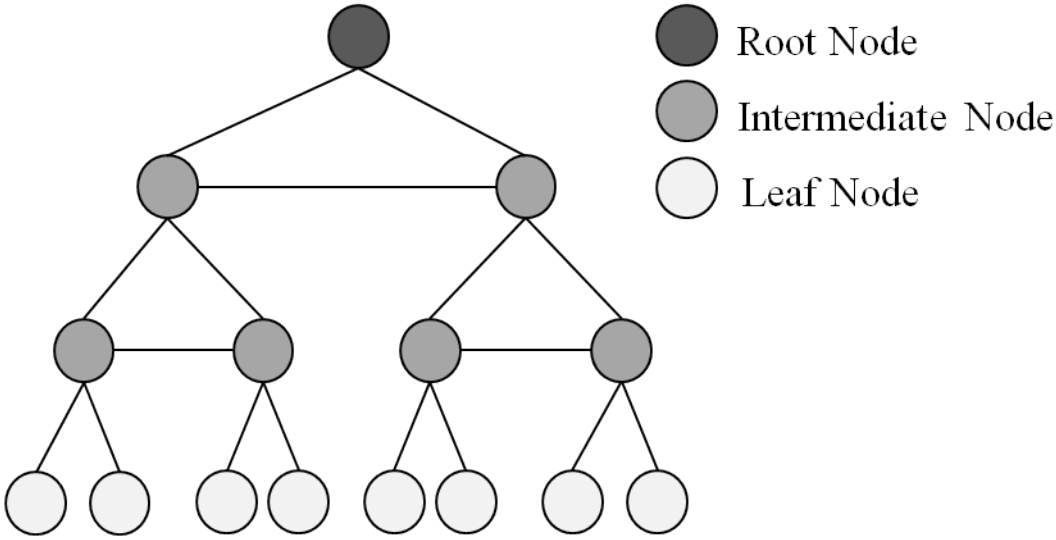

Figure 2.2: Sibling tree architecture

◇ **Graph like topology** is the sensible representation of a grid. Any node can be connected to any other node without any restrictions of tree topology. There is no central node designated as a root node, and any node can be connected with any number of nodes. In Figure 2.3, an example graph like grid architecture is given.

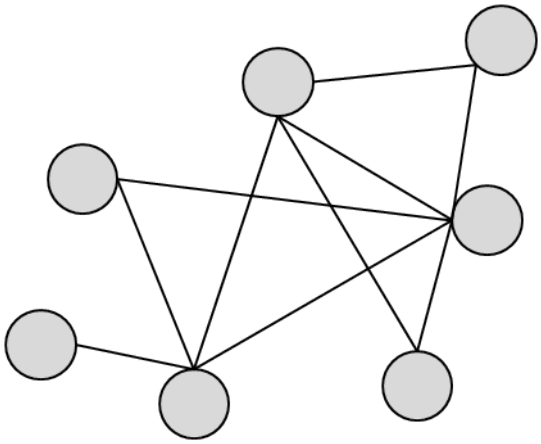

Figure 2.3: Graph-like architecture

◇ **Peer to peer topology** is a specific type of graph like topology. Each node is connected to other nodes in the grid and without the need of a

central node, nodes can meet the needs of the grid. It is also extensible, hence adding new nodes does not damage grid topology. Figure 2.4 shows a peer to peer grid architecture.

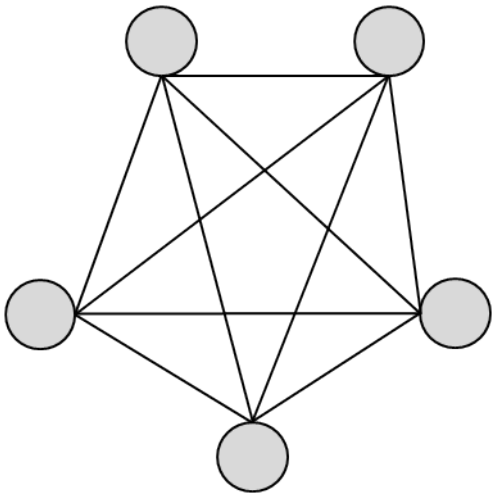

Figure 2.4: Peer to peer grid architecture

◇ **Hybrid model** is any combination of provided topologies.

2.3 Data Replication

In data intensive applications, datasets must be shared by a community of hundreds or thousands of researchers distributed worldwide. These researchers need to be able to transfer large subsets of these datasets to local sites or other remote resources for processing. Ensuring efficient access to such huge and widely distributed data is a serious challenge to network and grid designers. The major barrier to supporting fast data access in a grid is the high latencies of Wide Area Networks (WANs) and the Internet. Therefore, optimizing the use of available resources is an important challenge to undertake during the construction of data grids [11]. Optimization of data access can be achieved via *data replication*, where identical copies of data are generated and stored at various sites. This can significantly reduce data access latencies and network load; and maximize the data availability and fault tolerance [12].

Replica selection is the process of choosing a replica that will provide an application with data access characteristics that optimize a desired performance criterion, such as absolute performance, cost, or security. Replication decisions are made based on a cost model that evaluates data access costs and performance gains of creating each replica. The estimation of costs and gains is based on such factors as run-time accumulated read/write statistics, response time, bandwidth, and replica size [8, 9].

Two kinds of replication methods are possible as static and dynamic based on manner of working:

- **Static Replication** [13]: Static replication is an "off-line" process whereby replicas are placed using a snapshot of the system at design time. The replication sites are chosen before the system comes "on-line" and the sites chosen will continue to store replicas even if the system changes significantly. In a static replication strategy, the number of replicas and the host node is chosen statically at the start of the life cycle; no more replicas are created or migrated after that. Two approaches are used to solve the static data optimization problem:
  - *Integer Programming*
  - *Simplification*
- **Dynamic Replication**: The dynamic replication changes the location of replicas and creates new replicas to adapt to changes in user request pattern, storage capacity and bandwidth. Dynamic replication algorithms can create replicas on new nodes and can delete replicas that are no longer required depending on the global information of the data grid. Generally, replication decisions are based on the number of access (NOA) for each file. A typical dynamic replication algorithm breaks the time into sessions. At the beginning of each session, the replication algorithm is invoked to determine the replica placement based on the placement that is done in the previous session. The replica servers will be filled with replicas in the long run and some replicas will be deleted to make room for new ones. Dynamic replication methods can be categorized as centralized and distributed:

– In *the centralized dynamic data replication methods* [2, 14, 15], there is a replication master running in the system. Each replica server collects the records of data accesses that are initiated by the computing sites in its domain. When it is time to replicate data, all replica servers send the collated historical information to the replication master. The replica master computes the values of NOA for each file. By utilizing NOA results along with other information about the grid, popular files are replicated. The files with larger NOA may be replicated more times than those with smaller NOA.

– In *the distributed dynamic data replication infrastructure* [16–20], for every data access request from a computing site, the replica server records the request into its history table. The historical records are periodically exchanged among all replica servers. Each replica server aggregates NOA over all domains for the same data file and creates the overall data access history of the system. At intervals, each replica server will use the replication algorithm to analyze the history and determine data replications.

2.4 Job Scheduling

In large-scale data-intensive applications, data transfer is the primary cause of job execution delay. The main task of a scheduler is to assign jobs to nodes based on certain criteria. The problem of scheduling an application composed of a set of independent tasks, each of which requires multiple data sets that may be replicated on multiple grid sites. Scheduling operation assigns the set of tasks to the selected grid sites.

Scheduling algorithms can be classified into two types as batch mode and on-line mode.

◇ **Batch Mode Job Scheduling Algorithms:** In the batch mode, the jobs are collected into a set during a specific duration, and this set of jobs is

scheduled at predetermined time periods. Some examples of this approach are given below [21]:

- In *First Come First Served scheduling algorithm (FCFS)* [22], jobs are executed according to the order of job arriving time. The next job which has the smallest arrival time will be executed in turn.
- In *Round Robin scheduling algorithm (RR)* [23], a fixed time quantum is defined. Each job can be executed only within this quantum. If the job cannot be completed in one quantum, it will return to the queue and wait for the next round.
- *Min-Min and Max-Min algorithm* [24] sets the job that has the earliest completion time with the highest priority. Each job is always assigned to the resource that can complete it earliest. Similar to Min-min algorithm, Max-min algorithm gives the highest priority to the job with the maximum earliest completion time.
- In *Sufferage scheduling algorithm* [25], each job is assigned according to its sufferage value. The sufferage value of a job is defined as the difference between its second earliest completion time and its earliest completion time. The sufferage algorithm will pick a job in an arbitrary order and assign it to the resource that gives the earliest completion time. If another job has the earliest completion time with same resource, the scheduler will compare their sufferage values and choose the larger one.

◇ **On-line Mode Heuristic Scheduling Algorithms:** In this approach, jobs are scheduled to grid sites as soon as they arrive. Some examples of this approach are given below:

- *The Shortest Turnaround Time (STT) heuristic* [26] estimates the turnaround time on every computing site and assigns the current job to the site that provides the shortest turnaround time.
- *The Least Relative Load (LRL) heuristic* [26] assigns the current job to the computing site that has the least relative load. This scheduling

heuristic attempts to balance the workloads for all computing sites in the data grid.

– *Data Present (DP) heuristic* [26] takes the data location as the major factor when making an assignment decision for a job.

2.5 The Integration of Job Scheduling and Data Replication

In data grids, both scheduling and replication aim at reducing the latency for job execution as explained in Section 2.3 and 2.4. These two techniques may be complementary with each other: performing job scheduling without data replication places an overhead of data transfer time as job's input data files have to be fetched remotely, while performing data replication without job scheduling does not result in effective utilization of data grid resources, as moving data costs more bandwidth and takes longer transfer time than moving jobs does. Therefore, integrating scheduling and replication to optimize the system performance in data grid has been an active research area.

The benefits of job scheduling and data replication strategies are as follows:

◇ Almost all scheduling and replication strategies try to reduce the access latency, thus reducing job response time and hence increasing performance of the data grids.

◇ Replication strategies improve the data availability.

◇ When replication improves availability, the reliability is improved as well. The more the number of replicas, the more the chance that users' requests will be serviced properly, and hence the more reliable the system is.

◇ Almost all the replication and scheduling strategies try to reduce the bandwidth consumption to improve the availability of data and performance of

the system. The aim is to keep the data as close to submitted jobs as possible, so that data can be accessed efficiently.

- ◇ Some of the scheduling and replication strategies target to provide a balanced workload on all data servers. This helps in increasing performance of the system and provides better response time.

- ◇ With a higher number of replicas in a system the cost of maintaining them becomes an overhead for the system. Some replication strategies aim to make only an optimal number of replicas in the data grid. This ensures that the storage is utilized in an optimal way and the cost of replica maintenance is kept low.

- ◇ Job execution time is another very important parameter. Some replication and scheduling strategies target to minimize the job execution time with optimal replica placement. The idea is to place the replicas closer to the jobs in order to minimize the response time, and thus job execution time. This increases the throughput of the system.

2.6 Graph Partitioning

An undirected graph is given as $G = (V, E)$, where V is the set of vertices and E is the set of edges, and the number of vertices in G is given by $|V| = n$. Every edge $e_{ij} \in E$ connects a pair of distinct vertices v_i and v_j. Two vertices v_i and v_j are adjacent if $e_{ij} \in G$. $Adj(v_i)$ denotes the set of vertices adjacent to v_i and the degree of v_i is equal to the number of edges incident to it, $d_i = |Adj(v_i)|$. If all vertices of G are pair wise adjacent, then G is a complete graph. Weight of vertex v_i is denoted as w_i and cost of edge e_{ij} is denoted as c_{ij}.

A graph is called *bipartite* if V admits a partition into two parts such that every edge has its ends in different parts: vertices in the same part must not be adjacent.

$\prod = \{V_1, V_2, ..., V_K\}$ is a *K-way partition* of G if the following conditions hold:

- ◇ Each part $V_k \in \prod$ is a nonempty subset of V for $1 \leq k \leq K$.
- ◇ Parts are pair wise disjoint.
- ◇ Union of K parts is equal to V.

Edges between vertices of different parts are called *cut (external)* edges and all other edges are called *uncut (internal)* edges. The *cutsize* of a partition is defined to be the sum of the weights of the edges that are cut. The objective of graph partitioning is to find a partition which balances the vertex weights in each subdomain and minimizes the cutsize. The weight of a part is the sum of the weights of the vertices that are in that part. A *balanced k-way partition* is a partitioning of the vertex set V into k disjoint subsets where the difference of cardinalities between the largest subset and the smallest one is at most one. For a $(k, 1+\in)$ balanced partition problem, the maximum number of vertices in each part is set to $max_i |V_i| \leq (1+\in)\frac{|V|}{k}$. To improve the quality of partitions, imbalance is allowed. Imbalance of a partition $\prod$ given in Formula 2.1 is the ratio between the maximum weight of a part and the average weight of parts.

$$max_{V_i \in \prod_k} \left(\sum_{v_j \in V_i} w_j \right) \leq Imbalance * \left\lceil \sum_{v_j \in V} \frac{w_j}{k} \right\rceil) \tag{2.1}$$

Some graph partitioning softwares are Chaco [27], Jostle [28], Metis [29], Party [30], Scotch [31].

2.7 Iterative Improvement Heuristics

Given a partition on a graph, the aim of iterative improvement algorithms for graph partitioning is to reduce the cutsize of a partition by moving or swapping vertices between parts. These algorithms try to improve the cost by a series of move or swap operations on vertices of partitions. Two widely used iterative improvement algorithms are:

- ◇ The Kernighan-Lin [32] is an iterative algorithm that starts with an initial bipartition of the graph and in each iteration it searches for a subset of vertices, from each part of the graph such that swapping them leads to a partition with smaller cutsize. If such subsets exist, then the swap is performed and this becomes the partition for the next iteration. The algorithm continues repeating this entire process until no improvement can be made.

- ◇ The Fidducia- Mattheyses [33] algorithm moves one vertex from one partition to the other in attempt to minimize the cutsize of the partition. Unlike KL heuristic that swaps pairs, this algorithm is based on move operations.

Chapter 3

Related Work

The related work can be viewed from three perspectives. The first is related to data replication, the second is related to job scheduling and the last part related to integrated data replication and job scheduling.

3.1 Data Replication in Data Grids

In [13], the authors study data replication on data grids as a static optimization problem. They show that this problem is NP-hard and non-approximable, which means that there is no polynomial algorithm that provides an approximation solution if $P \neq NP$. The authors discuss two solutions: integer programming and simplifications. The limitation of the static approach is that the replication cannot adjust itself to dynamically changing user access pattern. They provide a centralized integer programming technique.

A polynomial time, centralized, greedy data replication algorithm is proposed in [2]. The grid structure is modeled as a graph based architecture. Each file is stored in sites where it is originally produced and except their existence, all grid sites have empty storage space. Then, at each step, the algorithm replicates one data file into the storage space of one site such that the reduction of total

access cost in the data grid is maximized at that step. The algorithm terminates when all storage space of the sites has been exhausted-filled, or the total access cost cannot be reduced further. Experiment shows that the algorithm reduces the total data file access time (compared to no replication) at least half of that obtained from the optimal solution.

In [16, 34], the authors provide six dynamic replication strategies for a hierarchical (multi-tiered) data grid architecture: (1) No Replication: only the root node holds replicas; (2) Best Client: replica is created for the client who accesses the file the most; (3) Cascading: a replica is created on the path to the best client that accesses the file most; (4) Plain Caching: a local copy is stored upon initial request; (5) Caching plus Cascading: combines plain caching and cascading; (6) Fast Spread: file copies are stored at each node on the path to the best client. All of these strategies are evaluated with three different kinds of access patterns: (1) Random Access: there is no locality in access patterns; (2) Temporal Locality: recently accessed files are likely to be accessed again; (3) Geographical plus Temporal Locality: Files recently accessed by a site are likely to be accessed by nearby site. They show that the cascading strategy reduces response time when data access patterns contain both temporal and geographical locality. When the access pattern contains some locality, Fast Spread saves bandwidth over other strategies.

In [14], the authors present a dynamic replication algorithm for multi-tier data grids. They propose two dynamic replication algorithms: Single Bottom Up (SBU) and Aggregate Bottom Up (ABU). SBU algorithm replicates the data file that exceeds a pre-defined threshold for clients. Because SBU does not consider the relationship with historical access records, ABU is designed to aggregate the historical records to the upper tier until it reaches the root. With the hierarchical topology, the client searches for files from a client to the root. In addition, the root replicates the requested data at every node, so the access latency can be improved significantly. However, a lot of storage space will be wasted. Performance results show both algorithms reduce the average response time of data access compared to a static replication strategy in a multi-tier data grid.

The study [20] proposes a *dynamic, greedy, decentralized* data replication algorithm for *graph based grid architectures*. The algorithm tries to maximize the availability of data in data grid assuming limited replica storage space. It categorizes the data into two as hot and cold. Hot data is the data that is being used more frequently and it is treated differently than cold data while assigning weight measures to files for replacement. The emphasis is that the availability of the whole system is more important than the availability of a single file and the correctness of available data is of prime importance. Replication is performed at four steps. The algorithm first checks whether requested file is present in storage element. If it is present, then no replication is performed. If requested file is not present, then the optimizer checks free storage space available, and if it is large enough, requested file is replicated. Thirdly, if there is not enough space available, the optimizer has to select files to be removed to make enough room for the new file depending upon their weights. Finally, it has to guarantee that replication gain is more than replacement loss. Test results show that replica schemes perform better than the binomial economical replica scheme, zipf economical replica scheme [35, 36], LFU and LRU.

In [15], the authors propose a *dynamic centralized* data replication mechanism for *multi-tiered data grid architecture* called Latest Access Largest Weight (LALW). It is assumed that the popular files in the past will be accessed more than the others in the future. This is called temporal locality. With this property, a popular data file is determined by analyzing the number of access to the data files from users. After finding the most popular file, they trace the client that generated the most requests for the most popular data file and a new replica is placed in it. To determine which file should be replicated, histories of records about the end-to-end data transfers are collected. By using different weights for records, LALW determines which files are more popular, and hence should be replicated. The results show that the total job execution time of LALW is similar to LFU. However, LALW excels in terms of effective network usage and storage usage.

Economy based replication strategy [35] is provided as a long-term optimization technique. It aims to minimize the overall cost of file access on data grid

given a finite amount of storage resources. In this economy model, data files are regarded as the goods in the market and are traded by different grid site according to file requests from running jobs. When requesting a replica, a job will try to access the cheapest replica in the grid by starting an auction. Storage resources that have the file locally may reply by bidding a price that estimates the cost of data transfer. If the storage resource at a grid site is already filled up with replicas, selection and deletion of expendable file can create space for a newly requested data. Within the economy model, a prediction function is used to estimate the future revenues of data files.

In [19], the authors propose a *dynamic decentralized* data replication strategy, called BHR, which benefits from network-level locality to reduce data access time by avoiding network congestion in a data grid. In the proposed model, there can be different network regions combined with each other. If a required file is present within a region, there will be less number of routers in path, but if the file has to be fetched across another region, there will be more number of routes in the path. Within a region there will be broad bandwidth available. Network level locality means that if the required file is fetched from the site having broad bandwidth, it will reduce the response time significantly. BHR tries to improve the network level locality by replicating files within the region. If the required file is not present within the site, it is fetched from any other site, and it is decided whether to replicate this file or not. If the local storage element has enough space, the file is stored. If the available space is not enough, existing files are removed to make room for new files. The results of BHR are compared with LRU delete strategy and Delete Oldest strategy and show that BHR takes shorter total job time.

In study [18], authors present *dynamic decentralized* replication strategies based on utility and risk for two different kinds of access patterns. They use graphs to represent data grid. Before placing a replica at a site, they consider both utility and risk index for each site according to the current network load and user requests. A site with optimized utility and risk index is then chosen for replication. Their model uses average response time as a basis for comparison with various replication strategies. The replication strategies are based on distributed and decentralized model. Furthermore, they are dynamic so they can

adapt changes to both user and network behavior.

In [17], the authors propose a dynamic, *decentralized* data replication strategy for *peer to peer grid architecture*. In this approach, the peers can automatically produce the replicas whenever it is required to improve the availability of data.

In [37], the sharing of files among tasks is modeled as a hypergraph and employed hypergraph partitioning to obtain a computationally load-balanced mapping of tasks onto compute nodes that reduced remote I/O operations for file transfers. It is assumed that a compute node had enough disk space to hold all of the files staged on that node and replication of files is not considered.

3.2 Job Scheduling in Data Grids

Min-Min [38] is a well-known algorithm for job scheduling. When computing the expected minimum completion time (MCT) of a job on a node, Min-Min takes into account the files already available on the node and files already available on other compute nodes which can act as alternate sources for creating file replicas other than the remote storage system. When a job is scheduled on a node, all of its files are staged on the corresponding node. This leads to an implicit replication policy as multiple copies of files may be created on different nodes of the compute cluster. Each file required for a job is staged from one of the replicas or from the storage cluster such that the time to transfer the file is minimized.

In [21], a *hierarchical* framework and a job scheduling algorithm called Hierarchical Load Balanced Algorithm (HLBA) are proposed for grid environment. In the proposed algorithm, the system load is used as a parameter in determining a balance threshold. The scheduler changes the balance threshold dynamically when the system load changes. The main contributions of this paper are twofold: (i), the scheduling algorithm balances the system load with an adaptive threshold, and (ii), it aims to minimize the makespan of jobs.

3.3 Integrated Data Replication and Job Scheduling in Data Grids

In [39], *dynamic* data replication algorithms for *centralized*, *distributed* and *online mode* grid scheduling heuristics, Shortest Turnaround Time (STT), Least Relative Load (LRL) and Data Present (DP) are proposed. In this research, replication aims to shorten data access time perceived by the job. It is assumed that the popular data in the past phase remains popular in the near future. Thus, the dynamic data replication algorithms discussed in this paper determine the popular data by analyzing data access history. In the centralized data replication method, replica master uses a table that ranks each file access in descending order. If a file access is less than the average, it will be removed from the table. Then, it pops files from top and replicates them using a response-time and server-merit oriented replica placement algorithms. In the decentralized method, every site records file accesses in its table and exchange this table with its neighbors. Every domain knows average number of access for each file.Utilizing this information, they delete files whose accesses are less than the average, and replicates other files in its local storage. The grid scheduling heuristics studied in this paper are online mode heuristics as STT, LRL and DP. For each incoming job, STT assigns the job to the site that provides the shortest turnaround time, LRL assigns the job to the site that has the least relative load, and DP assigns the job to the site that has its input files. Experiments show that DP scheduling heuristic works better than STT and LRL and centralized replication method performs better than distributed replication method. When integrating scheduling and replication, STT + CDR exhibits the best performance among others.

In [5], the authors develop a family of job scheduling and replication algorithms and use simulation studies to evaluate them in *multi-tiered grid architectures*. Three different replica placement algorithms are considered: (1) *Data-DoNothing*: no active replication; (2) *DataRandom*: a replica is created at random site when request for a particular file exceeds a particular threshold; and (3) *DataLeastLoaded*: a replica is created at the site with the smallest number of waiting jobs. These three replication strategies are combined with four

scheduling strategies: (1) *JobRandom*: jobs are scheduled to random sites; (2) *JobLeastLoaded*: jobs are scheduled to the site with fewest waiting jobs; (3) *JobDataPresent*: jobs are scheduled to the sites containing the required data and with the fewest waiting jobs; (4) *JobLocal*: jobs are scheduled locally. The results suggest that while it is necessary to consider the impact of replication on the scheduling strategy, it is not always necessary to couple data movement and scheduling. Instead, these two activities can be addressed separately, thus significantly simplifying the design and implementation of the overall data grid system. They show that when there is no replication, simple local scheduling performs the best. However, when a replication schema is used, scheduling jobs to the sites containing the required data is a better approach.

The work in [40] has detailed a job scheduling and file replication mechanism for a batch of data intensive jobs that exhibit batch-shared I/O behavior. Batch shared I/O behavior means the same file is the input of multiple jobs in a batch. They propose a 0-1 Integer Programming based approach that couples scheduling and replication and a BiPartition heuristic that decouples scheduling and replication. The performance results show that their strategies perform better than Min-Min [24], JobDataPresent [5]. The experimental results show that: 1) the IP scheme achieves the best batch execution time, but has significant scheduling overhead, thereby restricting its application to small scale workloads, and 2) the BiPartition scheme is a better fit for larger workloads and systems - it has very low scheduling overhead and no more than 5-10% degradation in solution quality, when compared with the IP-based approach.

The work in [41] deals with the problem of integrating job scheduling and data replication strategies, called Integrated Replication and Scheduling Strategy (IRS). It decouples job scheduling from data scheduling. At the end of periodic intervals when jobs are scheduled, the popularity of files is calculated and then used by the data scheduler to replicate data for the next set of jobs, which may or may not share the same data requirements as the previous set.

The research [42] proposes a framework that integrates scheduling, replica placement optimization and replication strategies to optimize the performance of

the data grid system. It is assumed that the data transfer is the primary cause of job execution delay. The proposed scheduler finds the best computing node with minimum execution time for executing a submitted job and dispatches the job to that node for execution. Using statistics collected from the network and resource monitoring services, which is a view of the grid resources' status, the data and replica management service pro-actively redistributes the datasets over the data grid sites with the goal of minimizing total access cost. The information collected may include information such as data access patterns, locations and capacities of storage resources, user and application behaviors, bandwidth availability and latency, and the computation power of the compute nodes. The scheduler utilizes Tabu Search optimization heuristic to dispatch jobs to the compute nodes. The framework employs smart algorithms for the placement of the replicas and integrates it with traditional replication strategies, and couples them with intelligent scheduling of jobs.

Chapter 4

A Bipartite Graph Model and a Graph Partitioning Approach

4.1 A Graph Model for Data Grid Architecture

A data grid consists of a set of sites, and each site contains storage and computation units. The storage units provide storage for files, and each file is placed in a storage unit. To obtain improved performance and availability, the data files are usually replicated. Each file may have several replicas in the grid and each of them is stored in different storage units. The computation units offer computational resources for tasks.

Basically, a data grid can be represented as an undirected graph $G = (V, E)$ where the set of vertices V represents the sites (S_1, S_2, ..., S_K) in the data grid and the set of weighted edges E represents the bandwidth between sites. The bandwidth between site S_i and site S_j is represented as B_{ij} in bandwidth matrix B. The Figure 4.1 depicts the grid model.

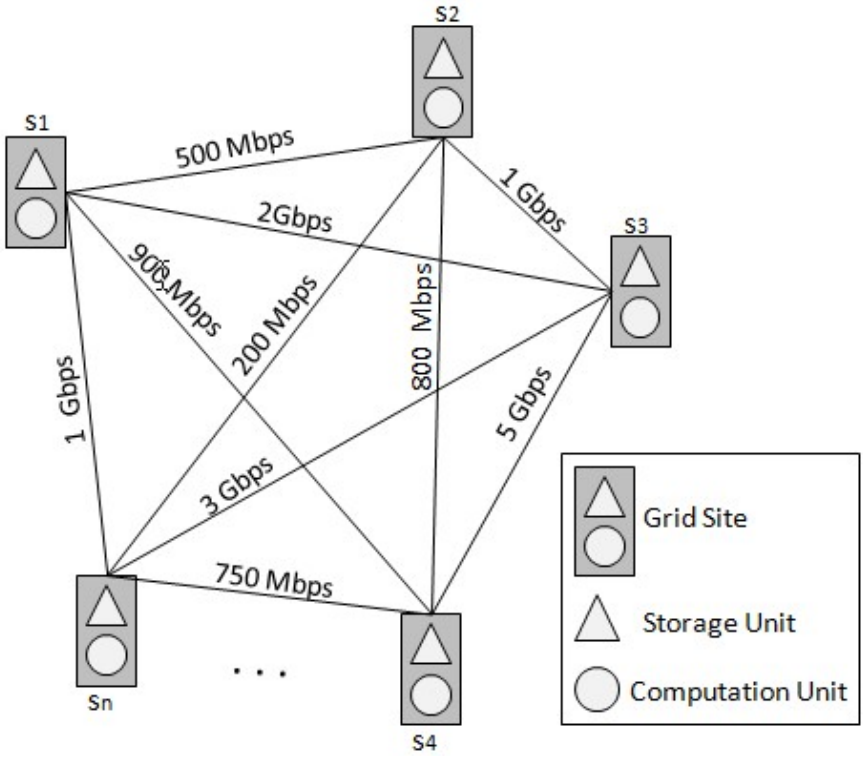

Figure 4.1: A Graph model for data grid architecture

While dealing with the *Part to Site Assignment* problem that will be discussed in Section 5.2, the shortest distance information will be needed. To compute file transfer times, we compute $D_{ij} = 1/B_{ij}$, which gives the transfer time of a file from site s_i to site s_j when multiplied by the size of that file. The more the bandwidth between sites is, the less the file transfer time between them turns out. This information is further utilized to compute shortest distances between sites using APSP as seen in Algorithm 1. The input to Algorithm 1 is the bandwidth information (B) and the number of sites ($|V| = K$). The output is D matrix that provides us to compute the shortest transfer times of files between sites. Note that the bandwidth values need not to form a complete graph. The complexity of Algorithm 1 is $O(K^3)$ which is dominated by the complexity of APSP.

Algorithm 1: CONSTRUCT-SHORTEST-DISTANCE-MATRIX

Input: An integer $N = 1, 2, ...K$ where K is the number of sites.
Input: An $N \times N$ bandwidth matrix
▷ All edge weights are in a common unit like Mbps. Bandwidth information is converted distance information.

```
foreach i ∈ (N × N) do
    D_ij = 1/B_ij
D = ALL − PAIRS − SHORTEST − PATH(D)
Return D
```

4.2 A Bipartite Graph Model for Tasks and Files

A bipartite graph can be used to represent the relationship between tasks and files in the grid system. There are n data files $F = f_1, f_2, ..., f_n$ and m tasks $T = t_1, t_2, ..., t_m$ in the grid system. From this point on, we will use task and job interchangeably. Each task t_j needs a subset of data files $Adj(j)$ being its input files for execution. Figure 4.2 represents our bipartite graph model.

Let $U(x, y, \Delta)$ be a number sampled from the uniform distribution with a range from x to y, where the sampling granularity is Δ. File sizes are exactly set to $U(500MB, 5GB, 500)$, and the number of files that each task requires is set to $U(1, 10, 1)$. To simulate the file popularity in the grid system, *Zipf-like distribution* is used. In Zipf-like distribution, the number of requests for the nth most popular file is directly proportional to n^{α}, where α is a constant. The observed parameter values are in the range of $0.65 < \alpha < 1.24$ [43, 44].

The vertices of the graph stand for both files and tasks. An edge between file f_i and task t_j indicates t_j requests f_i. We construct a two-constraint vertex weight structure for vertices to distinguish between task and file vertices. The weight of a file vertex is set to the size of the file, and the weight of a task vertex is set to the summation of size of the files that are accessed by the task. The size of a file f_i is represented as s_i. The equations for weights of vertices are given below respectively:

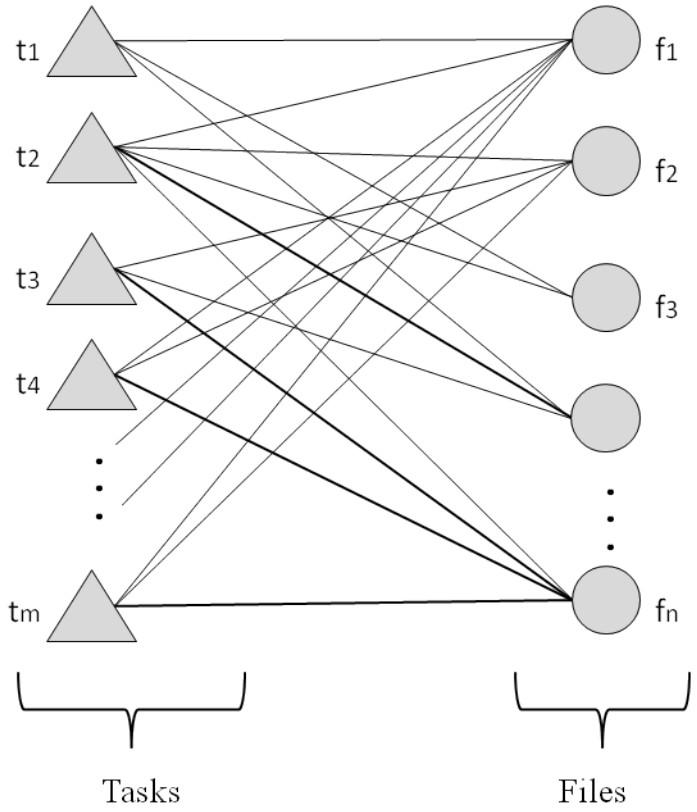

Figure 4.2: A bipartite graph model for tasks and files

◇ For task vertices: $\begin{cases} w_1(t_j) = \sum_{f_i \in fs_j} s(f_i) \\ w_2(t_j) = 0 \end{cases}$

◇ For file vertices: $\begin{cases} w_1(f_i) = 0 \\ w_2(f_i) = s(f_i) \end{cases}$

4.3 Partitioning of Bipartite Graph

We use multilevel *METIS Graph Partitioning Tool* [29] to partition task-file bipartite graph. Before partitioning, the most popular files shown in Figure 4.3 are removed to be replicated later. These files are requested by many tasks, and it makes partitioning process difficult. To find these files, the average number of accesses to files($favg$) is calculated. If a file is accessed more than the file average multiplied by a predetermined coefficient (λ), it is considered as a popular file

and needs to be replicated. After removing these files temporarily, the remaining graph is partitioned illustrated in Figure 4.4 into K parts ($V_1,\ V_2, ...V_K$) where K is the number of sites. As shown in Figure 4.5, each partition corresponds to a task-file set that will be assigned to a site. During this phase, the bandwidth and hop information between sites are not considered.

The objective of partitioning process is to minimize the number of messages sent between sites, disregarding the distance information and message sizes which will be handled in a later phase. Obtaining a balance on the first weight corresponds to balancing computational loads in a rigorous manner while obtaining a balance on the second weights corresponds to balancing storage of sites

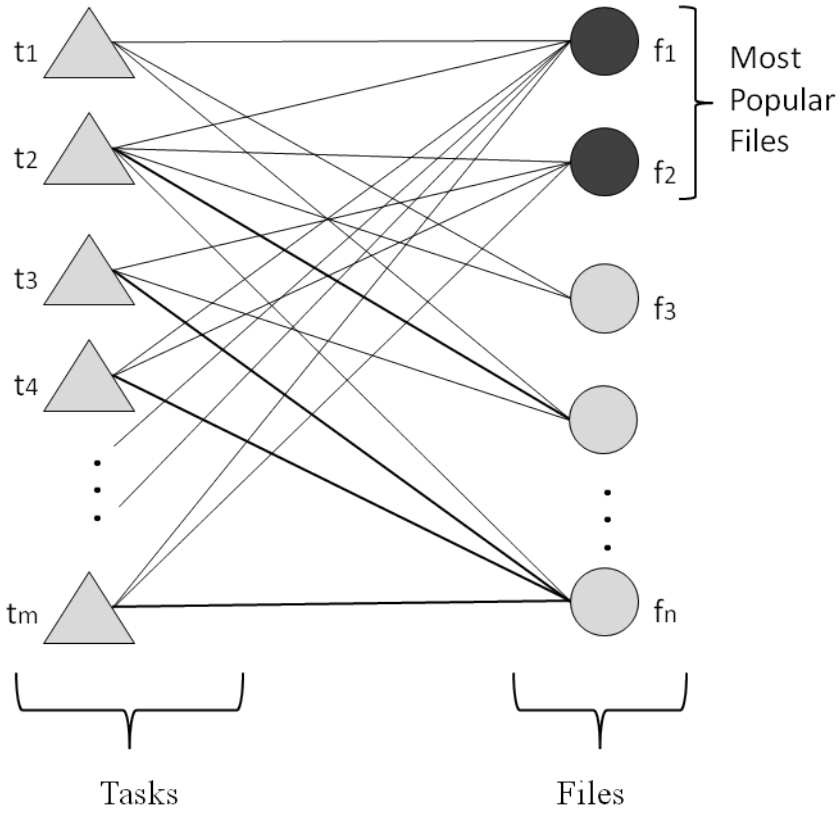

Figure 4.3: Representation of tasks and files

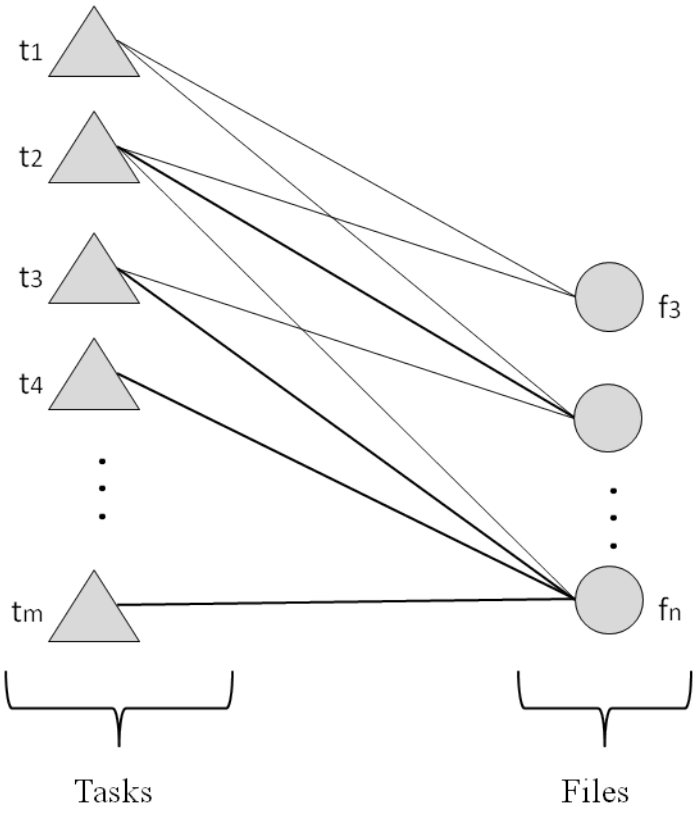

Figure 4.4: Tasks and files after removing popular files

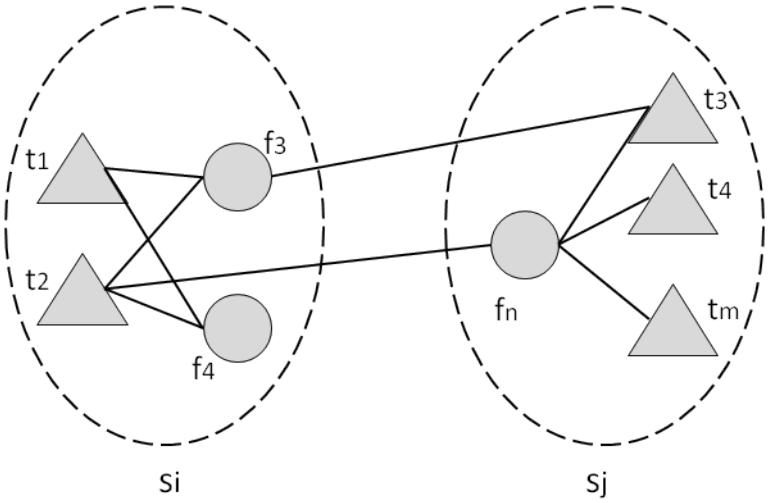

Figure 4.5: Partitioning of tasks and files

Chapter 5

Part to Site Mapping

In Section 4.2, a bipartite graph has been proposed for modeling tasks and data in the grid system using the access logs. Then, in Section 4.3, this graph has been partitioned to obtain a data placement and job scheduling strategy. In this chapter, the obtained parts are further refined in order to be mapped to grid sites as modeled in Section 4.1 by using a KL-based heuristic that takes the bandwidth and hop information between sites into account.

In this chapter, Section 5.1 presents the problem of part-to-site mapping. Section 5.2 describes the properties of KL-based Heuristic that we use to solve this problem. In Section 5.3, the Overall KL Algorithm is described.

5.1 Problem Definition

In part-to-site mapping problem, we are given a set of K parts ($\mathcal{V} = V_1, V_2, ...V_K$) obtained by graph partitioning and a set of K sites ($\mathcal{S} = S_1, S_2, ...S_K$) with bandwidth and distance information. The problem is to obtain a one-to-one mapping $f_M : \mathcal{V} \rightarrow \mathcal{S}$ between parts and sites. Any part can be assigned to any site, incurring some cost that may vary depending on the assignment made. There are $K!$ possible assignments. The cost of mapping part V_k to site S_m

$(f_M(V_k) = S_m)$ - in other words the communication cost - is defined

$$c_{km} = \sum_{\substack{t_j \in V_k \\ f_M(V_k)=S_m}} \sum_{\substack{f_i \in Adj(t_j) \\ f_i \in V_k \\ f_M(V_l)=S_n}} s(f_i) \times D_{mn}. \tag{5.1}$$

In the above formula, t_j is a task assigned to site S_m, $Adj(t_j)$ represents the subset of data files that t_j needs as its input files, $s(f_i)$ is the size of file f_i, and D_{mn} is the shortest distance between site S_m and site S_n where D is a symmetric matrix and computed as in Section 4.1. After the partitioning phase given in Section 4.3, each task $t_j \in f_M(V_k) = S_m$ needs some of its input files, namely $Adj(t_j)$, from other sites. To compute the cost of mapping V_k to S_m, we sum the multiplication of file size and the distance value for each file that is stored in sites other than the one in which the tasks of this site are stored. These values are computed and summed accordingly for all tasks that are assigned to the site being considered.

We utilize a transfer matrix T of size $K \times K$. The entry T_{kl} stands for the amount of files being transfered between part V_k and part V_l as the tasks in part V_k require some files from part V_l. Thus, T matrix indicates how much data transfer performed between parts. The diagonal entries of T matrix are equal to zero, and this matrix need not to be symmetric. Since the partitioning is done for once and stable, we can compute T matrix right after the graph partitioning phase. Each entry T_{kl} is computed as follows:

$$T_{kl} = \sum_{t_j \in V_k} \sum_{\substack{f_i \in Adj(t_j) \\ f_i \in V_l \neq V_k}} s(f_i). \tag{5.2}$$

By using T matrix, we can compute the cost by multiplying the distance value by the amount of file transfers for each site. By the help of T matrix, the cost of mapping part V_k to site S_m $(f_M(V_k) = S_m)$ can be computed as

$$c_{km} = \sum_{\substack{V_l \in \mathcal{V} \\ f_M(V_l)=S_n \neq S_k}} T_{kl} \times D_{mn}. \tag{5.3}$$

And the total cost is given by

$$\sum_{\substack{V_k \in \mathcal{V} \\ f_M(V_k)=S_m}} c_{km}. \tag{5.4}$$

The objective is to find a mapping f_M that minimizes Eq. 5.4.

This problem is actually similar to *quadratic assignment problem* [45]. In quadratic assignment problem (QAP), there exist a set of n facilities F and a set of n locations L. We are to allocate facilities to locations, depending on a cost model being a function of the distance and flow between the facilities. The objective is to assign each facility to a location such that the total cost is minimized. The problem is formulated by a flow matrix F and a distance matrix D. The objective function is as follows

$$min_{\phi \in S_n} \sum_{i=1}^{n} \sum_{i=1}^{n} F_{ij} \times D_{\phi(i),\phi(j)}. \tag{5.5}$$

where S_n is the set of all permutations $\phi : N \rightarrow N$. Each individual product $f_{ij} \times d_{phi(i)\phi(j)}$ is the cost of assigning facility i to location $\phi(i)$ and facility j to location $\phi(j)$. QAP is known to be NP_{hard} [45].

5.2 A KL-based Heuristic for Part to Site Mapping

In this section, we will elaborate on our algorithm for the initialization and update of swap gains, and Section 5.2.1 and Section 5.2.2 discuss these phases, respectively. We start with a random mapping of parts to sites, by which we calculate the initial gains accordingly. After this initiatory step, we try to improve the mapping via swapping parts.

5.2.1 Gain Initialization

In this section, we describe the initial gain computation algorithm in detail. At the beginning of the assignment and refinement process, the corresponding gain values of swapping any two sites need to be computed. In Algorithm 2, the initial gain computation algorithm is given. The algorithm takes K sites, distance information D and transfer sizes T as input and returns swap gains for each pair of sites.

Algorithm 2: INIT-GAINS-EXTENDED

Input: $\mathcal{V} = \{V_1, ...V_K\}$
Input: $\mathcal{S} = \{S_1, ...S_K\}$
Input: D, $K \times K$ distance matrix
Input: T, $K \times K$ transfer matrix
Input: $f_M : \mathcal{V} \rightarrow \mathcal{S}$, a mapping function

1 **foreach** *pair of parts* $(V_k, V_l) \in (\mathcal{V} \times \mathcal{V})$, $V_k \neq V_l$ **do**
  ▷ Let $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$
2 $g_{mn} \leftarrow 0$
  ▷ Compute the changes in the transfer cost of the sites that transfer files from sites $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$
3 **foreach** $V_x \in \mathcal{V} - \{V_k, V_l\}, f_M(V_x) = S_y$ **do**
4 $g_{mn} \leftarrow g_{mn} + T_{xk}(D_{ym} - D_{yn})$ ▷ Site S_m
5 $g_{mn} \leftarrow g_{mn} + T_{xl}(D_{yn} - D_{ym})$ ▷ Site S_n
  ▷ Compute the changes in the transfer cost of the sites $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$ transferring files from other sites
6 **foreach** $V_x \in \mathcal{V} - \{V_k, V_l\}, f_M(V_x) = S_y$ **do**
7 $g_{mn} \leftarrow g_{mn} + T_{kx}(D_{my} - D_{ny})$ ▷ Site S_m
8 $g_{mn} \leftarrow g_{mn} + T_{lx}(D_{ny} - D_{my})$ ▷ Site S_n

9 **return** g_{mn} for $1 \leq m, n \leq K$

In Algorithm 2, the for loop (lines 1-8) computes the gain values of swapping each pair of sites. Note that the gains obtained by swapping sites $S_m \leftrightarrow S_n$ and

$S_n \leftrightarrow S_m$ are equal, i.e. $g_{mn} = g_{nm}$. At the very beginning, the gain value is reset for each and every gain computation g_{mn}. Once we swap parts V_k ($f_M(V_k) = S_m$) and V_l ($f_M(V_l) = S_n$), their distances to the rest of all sites in the grid are interchanged. Thus, the transfer cost is also affected as the distances to the swapped sites are changed. In this case, we have to consider two distinct cases since the swap operation affects the transfer costs both from these sites and to these sites. Thereby, the gain computation is divided into two parts, namely two respective for loops. The first loop (lines 3-5) updates the gain value by adding the transfer cost of sites that transfer files from site S_m and site S_n. The second loop (lines 6-8) updates the gain value by adding the transfer cost of site S_m and site S_n that transfer files from other sites. The computed gain values are returned at the end of the algorithm (line 9).

The g_{mn} obtained by swapping parts V_k ($f_M(V_k) = S_m$) and V_l ($f_M(V_l) = S_n$) in Algorithm 2 is given by the following equation:

$$
\begin{aligned}
g_{mn} = & \sum_{\substack{V_x \in \mathcal{V} - \{V_k, V_l\} \\ f_M(V_x) = S_y}} T_{xk}(D_{ym} - D_{yn}) + T_{xl}(D_{yn} - D_{ym}) + \\
& \sum_{\substack{V_x \in \mathcal{V} - \{V_k, V_l\} \\ f_M(V_x) = S_y}} T_{kx}(D_{my} - D_{ny}) + T_{lx}(D_{ny} - D_{my}) \\
= & \sum_{\substack{V_x \in \mathcal{V} - \{V_k, V_l\} \\ f_M(V_x) = S_y}} (D_{ym} - D_{yn})(T_{xk} + T_{kx}) + (D_{yn} - D_{ym})(T_{xl} + T_{lx}) \\
= & \sum_{\substack{V_x \in \mathcal{V} - \{V_k, V_l\} \\ f_M(V_x) = S_y}} (D_{ym} - D_{yn})(T_{xk} - T_{xl} + T_{kx} - T_{lx})
\end{aligned}
$$

The Algorithm 3 does exactly the same thing with the Algorithm 2. This can further be improved by using a U matrix instead of T matrix where $U_{kl} = U_{lk} = T_{kl} + T_{lk}$. This final result is utilized in Algorithm 3.

As seen in Algorithm 3, for loop (lines 1-4) computes the gain values of swapping each pair of sites. As formulated in the above equation, we can combine two inner loops given in Algorithm 2 and obtain the loop in lines 3-4 in the latter algorithm.

Algorithm 3: INIT-GAINS-LITE

Input: $\mathcal{V} = \{V_1, ...V_K\}$
Input: $\mathcal{S} = \{S_1, ...S_K\}$
Input: D, $K \times K$ distance matrix
Input: T, $K \times K$ transfer matrix
Input: $f_M : \mathcal{V} \to \mathcal{S}$, a mapping function
foreach $(V_k, V_l) \in (V \times V), V_k \neq V_l$ **do**
    ▷ Let $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$
    $g_{mn} \leftarrow 0$
    foreach $V_x \in \mathcal{V} - \{V_k, V_l\}, f_M(V_x) = S_y$ **do**
        $g_{mn} \leftarrow g_{mn} + (D_{my} - D_{ny})(T_{xk} - T_{xl} + T_{kx} + T_{lx})$
return g_{mn} for $1 \leq m, n \leq K$

5.2.2 Gain Update

Performing a swap operation on two chosen sites S_m and S_n whose gain value is $g_m n$, the remaining gain values need to be updated to reflect the new mapping. After swapping $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$, transfer amounts of other sites from S_m and S_n are relocated and the distances between interchanged sites and other remaining sites are also changed. Consider part V_a ($f_M(V_a) = S_c$) needs to transfer T_{ak} amount of data from part V_k ($f_M(V_k) = S_m$) whose distance is D_{cm} and T_{al} amount of data from part V_l ($f_M(V_l) = S_n$) whose distance is D_{cn}. After swapping S_m and S_n, V_a has to transfer T_{ak} from S_n (with distance changed from D_{cm} to D_{cn}) and T_{al} from S_n (with distance changed from D_{cn} to D_{cm}). Furthermore, V_k needs to transfer T_{ka} with distance changed from D_{cm} to D_{cn}. Similarly, V_l needs to transfer T_{la} with distance change from D_{cn} to D_{cm}. The difference caused by swapping S_m and S_n is $T_{ak}(D_{cn} - D_{cm}) + T_{al}(D_{cm} - D_{cn}) + T_{ka}(D_{cn} - D_{cm}) + T_{la}(D_{cm} - D_{cn})$ and this value is reflected to gain value of swapping $f_M(V_a) = S_c$ with other sites. Assume we are to swap two sites $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$, and we are to update the gain value of swapping $f_M(V_a) = S_c$ and $f_M(V_b) = S_d$, g_{cd}. The gain value of swapping sites S_c and S_d before the actual swap operation $S_m \leftrightarrow S_n$ is given by:

$$g_{cd} = ... + T_{ka}(D_{md} - D_{mc}) + T_{kb}(D_{mc} - D_{md}) + T_{ak}(D_{md} - D_{mc}) + T_{bk}(D_{mc} - D_{md}) + T_{la}(D_{nd} - D_{nc}) + T_{lb}(D_{nc} - D_{nd}) + T_{al}(D_{nd} - D_{nc}) + T_{bl}(D_{nc} - D_{nd}) + ...$$

After swapping S_m and S_n, we need to update g_{cd} since distances are changed. Using T matrix without updating, we can compute new value of d_{cd} as follows:

$$g'_{cd} = ... + T_{ka}\,(D_{nd} - D_{nc}) + T_{kb}\,(D_{nc} - D_{nd}) + T_{ak}\,(D_{nd} - D_{nc}) + T_{bk}\,(D_{nc} - D_{nd}) + \\ T_{la}\,(D_{md} - D_{mc}) + T_{lb}\,(D_{mc} - D_{md}) + T_{al}\,(D_{md} - D_{mc}) + T_{bl}\,(D_{mc} - D_{md}) + ...$$

In order not to compute the gain values from the beginning, we can update them by considering the difference between g_{cd} and g'_{cd} as represented below:

$$g'_{cd} - g_{cd} = ... + T_{ka}\,(D_{nd} - D_{nc}) + T_{kb}\,(D_{nc} - D_{nd}) + T_{ak}\,(D_{nd} - D_{nc}) + T_{bk}\,(D_{nc} - D_{nd}) + \\ T_{la}\,(D_{md} - D_{mc}) + T_{lb}\,(D_{mc} - D_{md}) + T_{al}\,(D_{md} - D_{mc}) + T_{bl}\,(D_{mc} - D_{md}) + ...) - \\ (... + T_{ka}\,(D_{md} - D_{mc}) + T_{kb}\,(D_{mc} - D_{md}) + T_{ak}\,(D_{md} - D_{mc}) + T_{bk}\,(D_{mc} - D_{md}) + \\ T_{la}\,(D_{nd} - D_{nc}) + T_{lb}\,(D_{nc} - D_{nd}) + T_{al}\,(D_{nd} - D_{nc}) + T_{bl}\,(D_{nc} - D_{nd}) + ...)$$

$$= T_{ka}(D_{dn} - D_{cn} - D_{dm} + D_{cm}) + T_{ak}(D_{dn} - D_{cn} - D_{dm} + D_{cm}) + \\ T_{kb}(D_{cn} - D_{dn} - D_{cm} + D_{dm}) + T_{bk}(D_{cn} - D_{dn} - D_{cm} + D_{dm}) + \\ T_{la}(D_{dm} - D_{cm} - D_{dn} + D_{cn}) + T_{al}(D_{dm} - D_{cm} - D_{dn} + D_{cn}) + \\ T_{lb}(D_{cm} - D_{dm} - D_{cn} + D_{dn}) + T_{bl}(D_{cm} - D_{dm} - D_{cn} + D_{dn})$$

$$= (T_{ka} + T_{ak} + T_{lb} + T_{bl})(D_{dn} - D_{cn} - D_{dm} + D_{cm}) + \\ (T_{kb} + T_{bk} + T_{la} + T_{al})(D_{cn} - D_{dn} - D_{cm} + D_{dm})$$

$$g'_{cd} - g_{cd} = dif = (T_{ka}+T_{ak}+T_{lb}+T_{bl}-(T_{kb}+T_{bk}+T_{la}+T_{al}))*(D_{dn}-D_{cn}-D_{dm}+D_{cm}) \tag{5.6}$$

Hence, we can compute the new gain value g'_{cd} after swapping S_m and S_n by adding dif value in Eq. 5.6 old gain value g_{cd}. Thus, a single gain update operation can be performed in constant amount of time.

5.3 Overall KL Algorithm

In this section, we explain the refinement algorithm for part to site mapping in detail. In Section 4.3, we partition the set of tasks and files into K parts where

K is the number of sites in the data grid. The refinement heuristic starts from a random mapping of these K parts to K sites. We try to improve the quality of this mapping by successive swap of pair of sites with the aim of decreasing cost of the mapping. We start by choosing a pair of sites as S_m and S_n that provides the maximum swap gain value (g_{mn}). After interchanging these two sites, we update the gain values of remaining swap operations. After swap operation, we exclude all gains related to sites S_m and S_n. When all gain values are exhausted or no more improvement can be obtained on the cost, the algorithm terminates, and a rollback operation is performed to the point where the best cost is encountered.

In Algorithm 4, we show this refinement process. The inputs to Algorithm 4 are the parts $\mathcal{V}$, the sites $\mathcal{S}$, distance information D, transfer matrix T, a mapping function f_M and a parameter τ. The τ indicates the number of tolerated swaps that does not improve the cost.

As seen in Algorithm 4, we first compute the initial cost that we try to minimize through the algorithm (lines 1-3), the list of initial gain values, and we select the max gain represented as g^*_{mn} that is the swap of sites S_m and S_n (lines 5-6). This swap gives us earnings from the initial cost as $cost' = cost - G_{ij}$ Basically, the while loop (lines 7-25) is responsible for updating the remaining gain values. The for loop between lines 17 - 21 does the gain update part. The difference of the gain value is computed according to equation 5.6 and added to initial gain value. During update part, the new max gain g^*_{mn} is also found. After recomputing the gain values with unchanged T matrix, we swap T matrix according to old max gain. At the end of each iteration, sites of two parts are swapped ($f_M(V_k) = S_n \quad f_M(V_l) = S_m$). The algorithm terminates when there remains no gain in the list or no successful swap is found after carrying τ number of gains that satisfy no improvement. In order to control the latter situation, we check the decrease in the cost at each iteration (lines 8 - 15) and keep the count of gains that do not satisfy the condition. When the size of the list is equal to τ or all gains in G are exhausted, the refinement phase ends and rollback is applied for the unprogressive gains, and all swap operations associated with these gains are undone.

Algorithm 4: KL-REFINE-MAPPING

Input: $\mathcal{V} = \{V_1, ...V_K\}$, $\mathcal{S} = \{S_1, ...S_K\}$, D, T, $f_M : \mathcal{V} \to \mathcal{S}$, τ

▷ Compute initial cost.

$cost \leftarrow 0$

foreach *pair of parts* $(V_k, V_l) \in (\mathcal{V} \times \mathcal{V})$, $V_k \neq V_l$ **do**

  $cost \leftarrow cost + (T_{kl} + T_{lk}) \times D_{f_M(V_k)f_M(V_l)}$

$bestcost \leftarrow cost$

$G = \text{INIT-GAINS-LITE}(\mathcal{V}, \mathcal{S}, D, T, f_M)$

$g^*_{mn} = \max_{\substack{1 \leq i,j \leq K \\ g_{ij} \in G}} g_{ij}$ such that $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$

while $G \neq \emptyset$ *AND* $flagImproved = TRUE$ **do**

  $cost \leftarrow cost - g^*_{mn}$

  if $cost < bestcost$ **then**

    $bestcost \leftarrow cost$

    $count \leftarrow 0$

  else

    $count \leftarrow count + 1$

    if $count = \tau$ **then**

      $flagImproved = FALSE$

  $G \leftarrow G - \{g^*_{mn}\}$

  foreach $g_{xy} \in G$ **do**

    if $x = m$ *OR* $x = n$ *OR* $y = m$ *OR* $y = n$ **then**

      $G \leftarrow G - \{g_{xy}\}$

    else

      Update g_{xy} in constant time using Equation 5.6.

  Update T to reflect the swap of parts V_k and V_l.

  $f_M(V_k) = S_n$

  $f_M(V_l) = S_m$

  $g^*_{mn} = \max_{\substack{1 \leq i,j \leq K \\ g_{ij} \in G}} g_{ij}$ such that $f_M(V_k) = S_m$ and $f_M(V_l) = S_n$

Rollback to the point where the mapping with the best cost is encountered.

Chapter 6

Experimental Results

6.1 Data Grid Environment / Dataset Generation

In order to evaluate the performance of the graph partitioning method that we have employed and our KL-based heuristic for improving the quality of partitions, we have manually constructed a data grid environment using different parameters.

- ◇ *Files:* We vary the number of data files (—F—) as 1000, 2000, 5000, 10000. $U(x,\ y,\ \Delta)$ defines a number sampled from the uniform distribution with a range from x to y and the sampling granularity is Δ. The *size of files* is sampled from $U(500MB,\ 5GB,\ 100MB)$.

- ◇ We used Zipf-like distribution [43] (with parameter α) to determine file popularity. According to Zipf-like distribution, the access frequency of the j^{th} file is represented as $p_j = \frac{1}{j^\alpha \sum_{i=1}^{n} 1/h^\alpha}$ where n is the number of files. When $\alpha = 0$, the distribution follows uniform distribution. As α is increased, the distribution converges to strict Zipf distribution. In the experiments, we vary α as 0.6, 0.8, 1.0, 1.2.

- ◇ *Tasks:* We set the number of tasks ($|T|$) with a ratio according to the

number of files. Thus, we varied the number of tasks as $\frac{1}{2}$, 1, 2, 5, 10 times of number of files for each file set. Each task requests $U(1,\ 10,\ 1)$ number of files as its input data.

- ◇ *Sites:* We vary the number of grid sites (K) as $32, 64, 128, 256$ that for each K value, and we experiment all task-file sets to analyze the partitioning and improvement results vary when we fix the number of files and tasks and change K, and also when we fix K and change the number of files and tasks in the data grid.

- ◇ *Bandwidth:* It is assumed that each site has a link to all other sites, and the bandwidth capacity between sites is set to $U(500,\ 5000,\ 500)Mbps$. However, our approach can be generalized to any connector pattern between sites.

6.2 Partitioning Results

We utilized the METIS graph partitioning tool [29] to partition files and tasks into K parts. During this phase, we analyzed the partitioning quality considering *edgecut* metric that is the number of edges that are between different parts. That is the number of edges $(u,\ v)$ for which $P[v] \neq P[u]$. We conducted our experiments in three categories . Firstly, we investigated the impact of Zipf distribution on the partitioning quality. Secondly, we observed the effect of the file popularity threshold (λ) that is the deciding factor for having a somewhat threshold value prior to removal of files stated in 4.3. Lastly, we evaluated the impact of the ratio between the number of tasks and the number of files.

6.2.1 Partitioning Results based on the Zipf Values

We used the Zipf distibution in order to determine file popularity. We varied the Zipf values while fixing the file popularity threshold (λ) for file removal which will be replicated to all sites afterwards. The files that are requested more than

the average number of requests for each file multiplied by the λ will be replicated. We altered zipfian threshold α as $0.6,\ 0.8,\ 1.0,\ 1.2$ while fixing $\lambda = 1.5$. Table 6.1 displays the effects of zipf values on partitioning. We fixed the number of files as $|F| = 1000$ and presented three charts for three different values of the number of tasks as $|F| = 500,\ 2000,\ 10000$ given in Figure 6.1, Figure 6.2, Figure 6.3. The other figures are omitted since they have similar characteristics.

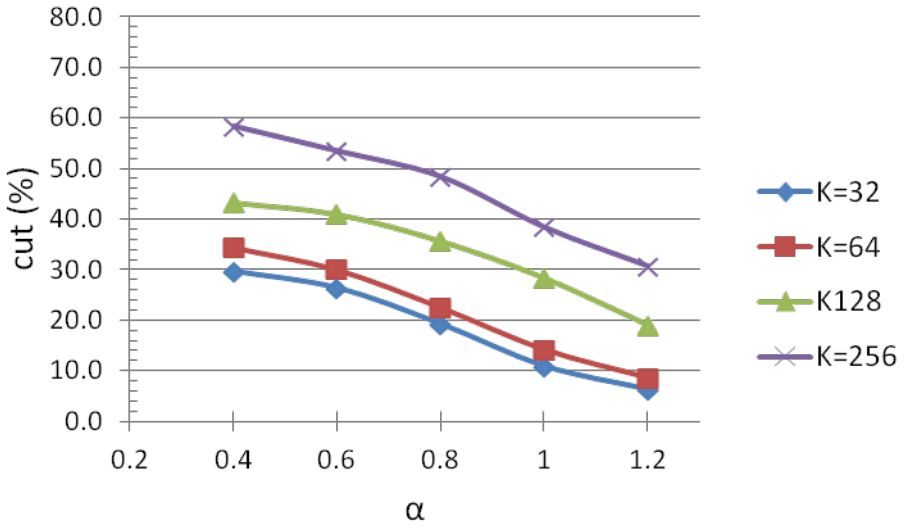

Figure 6.1: Cut (%) vs zipf value α ($|F| = 1000,\ |T| = 500,\ \lambda = 1.5$)

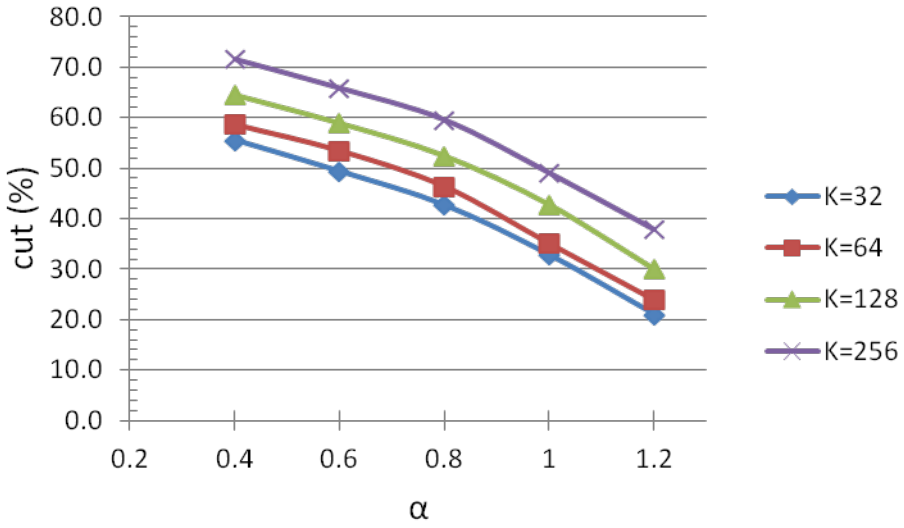

Figure 6.2: Cut (%) vs zipf value α ($|F| = 1000,\ |T| = 2000,\ \lambda = 1.5$)

		$\lvert F\rvert =$	$1K$					$2K$					$5K$					$10K$				
	zipf	$\lvert T\rvert =$	$0.5K$	$1K$	$2K$	$5K$	$10K$	$1K$	$2K$	$4K$	$10K$	$20K$	$2.5K$	$5K$	$10K$	$25K$	$50K$	$5K$	$10K$	$20K$	$50K$	$100K$
	0.6		24.6	39.9	49.2	57.4	62.4	23.6	38.8	49.2	56.9	61.7	21.8	38.6	48.7	56.7	61.3	21.9	38.4	48.8	56.9	61.1
$K = 32$	0.8		20.3	32.5	42.3	51.3	56.3	17.9	31	40.5	50	54.6	14.6	30	40.6	49.5	53.8	15.9	28.9	40.1	48.8	53.7
	1.0		7.1	23.2	32.7	43	46.6	9	19.4	28.9	39.1	44.4	6.9	18.4	27.7	36.6	41.5	6	16	25.5	35.5	40
	1.2		6.7	14.1	20.4	29.3	35.2	3.1	8.2	17.2	25.4	31.2	1.3	5.8	12.5	20.6	26.4	0.5	4.1	9.9	18.2	23.2
	0.6		28.5	43.7	53.2	61.8	65.9	27.3	42.2	51.3	60.5	65	24	40.7	51	59.7	64	23.5	40.9	51.2	5 9.5	63.9
$K = 64$	0.8		24.1	36.1	46.1	55.6	59.6	20.3	33.9	43.9	52.7	57.9	16.4	31.8	43	51.8	56.3	16.6	30.7	42.2	51.1	56
	1.0		10	26.9	36.3	45.8	49.7	10.7	21.9	31.7	41.8	46.8	8.1	20.1	29.4	38.8	43.4	6.6	17	27	37.6	42.1
	1.2		9.5	17.1	23.1	32.5	39.3	4.5	9.9	19.2	27.5	33.1	1.8	6.9	14.1	21.9	27.6	0.8	4.6	10.8	19.2	24.7
	0.6		33.4	48.4	56.8	63.7	68.1	30.8	45.5	54.2	63.4	67.5	27.2	43	53	61.9	66.1	25.4	42.2	53	61.7	65.6
$K = 128$	0.8		29.4	41.1	49.7	57.6	61.5	23.9	37.3	46.3	56	59.6	18.3	34.1	44.4	53.8	58.3	18.2	32.4	44.3	52.9	57.6
	1.0		15.7	30.9	39.7	48.4	52.1	13.9	24.3	34.4	43.6	48.3	9.6	22.1	31.2	39.8	44.5	7.6	18.3	28.5	39	43.3
	1.2		15.2	20.9	26.7	35.1	40.4	6.7	12.3	21.5	29.7	35.3	2.7	8	15.3	23.3	29.7	1.2	5.3	11.8	20.2	25.1
	0.6		39.9	54.2	61.2	66.5	69.5	35.1	49.2	57.5	64.9	68.9	29.2	45.6	55.5	64.6	67.6	27.4	44.2	54.7	63.3	67.1
$K = 256$	0.8		36.4	47.1	54.8	60.9	63.9	28.8	41.1	49.7	57.9	61.1	20.8	36.6	47	56	60	20	34.4	45.2	53.8	58.6
	1.0		21.4	37.7	45.3	51.2	53.4	19.3	28.4	37.5	46	50.7	11.9	24.4	33.5	42.4	47.1	8.9	19.9	29.7	39.8	44.3
	1.2		21	29.3	32	39.4	42.9	12.6	16.5	24.4	32	37.9	4.6	9.8	17.1	25	30.7	2	6.4	12.8	21.3	26

Table 6.1: Percentage of cut-edges for varying α values.

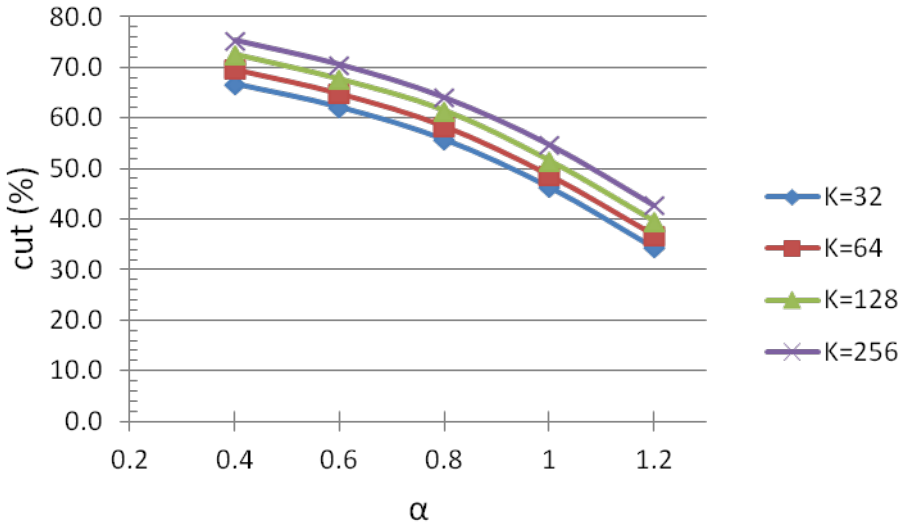

Figure 6.3: Cut (%) vs zipf value α ($|F| = 1000,\ |T| = 10000,\ \lambda = 1.5$)

As it is seen from Table 6.1 and Figs 6.1, 6.2, 6.3, the increase in the α values facilitates the decrease in the edgecut, and namely the improvement in the quality . As stated in [43], the increase in the Zipf values leads to distribution loosing uniform property. For this reason, the divergence from uniformity eventually makes some files being more demanded than the others in terms of the associated edges in the task-file graph representation. Since we remove the file vertices that have more edges than $\lambda \times favg$ to replicate later, the remaining task-file graph is partitioned in an easier way of good quality. This situation is also reflected in our results explicitly.

The actual trend or behavior shown in the graphs with respect to zipf and cut values are similar. In other words, the relation between zipf and cut values is inversely proportional. Similarly, when the value of K decreases, we have lower cut values which lead us to have better partitioning quality. This is basically because the probability of an edge being cut increases as K increases for fixed values of $\alpha,\ \lambda,\ |F|\ and\ |T|$.

6.2.2 Partitioning Results based on the File Popularity Threshold (λ)

We varied the λ values as 1.5, 3.0, 5.0, 10.0 while fixing α value as 1.0. In Table 6.2 and Figure 6.4, we give the partitioning results in terms of percentage of edges that are cut.

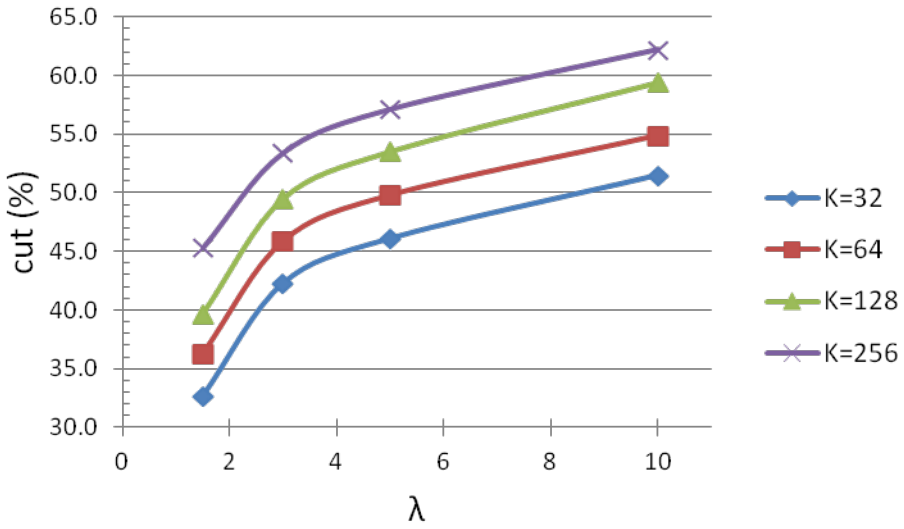

Figure 6.4: Cut (%) vs λ ($|F| = 1000, |T| = 2000, \alpha = 1.0$)

The results indicate that a higher (λ) value is a potential detriment to the quality of partitioning because of resulting in higher edgecut values. We have a so-called threshold value as $\lambda \times favg$ which determines the files to be removed for later replication. More explicitly, if we remove fewer files, the originality of the graph is nearly kept, and partitioning this graph gets harder which leads to higher cut values.

		$\lvert F\rvert =$	$1K$					$2K$					$5K$					$10K$				
	λ	$\lvert T\rvert =$	$0.5K$	$1K$	$2K$	$5K$	$10K$	$1K$	$2K$	$4K$	$10K$	$20K$	$2.5K$	$5K$	$10K$	$25K$	$50K$	$5K$	$10K$	$20K$	$50K$	$100K$
	NONE		42.3	57.7	65.3	70.8	74.1	42.4	55.5	65.1	70.7	73.2	40	57.4	65.9	70.3	73.3	41.2	57.2	64.6	70.2	72.9
	1.5		7.1	23.2	32.7	43	46.6	9	19.4	28.9	39.1	44.4	6.9	18.4	27.7	36.6	41.5	6	16	25.5	35.5	40
$K = 32$	3.0		16.9	32.5	42.3	49.6	54	16.3	29.5	38.2	46	51.7	13.3	26.4	36.4	44.4	48.6	13	24.1	34.1	43.3	47
	5.0		23.4	36.9	46.1	54.1	57.3	21.9	33.9	42.8	51.1	55.9	18.4	31.4	41.4	48.9	53.5	15.9	29.2	39.1	47.6	51.5
	10.0		28.5	43.9	51.5	57.9	62	27.2	39.3	48.7	55.8	60.2	21.8	37.2	46.7	53.8	58	20.9	34.6	44.3	52.2	56.3
	NONE		45.1	62.6	70	74.7	76.8	45.7	59.3	67.9	73.9	76.3	42.6	59.5	67.8	73.5	75.9	43.2	59.6	68.1	73.3	75.9
	1.5		10	26.9	36.3	45.8	49.7	10.7	21.9	31.7	41.8	46.8	8.1	20.1	29.4	38.8	43.4	6.6	17	27	37.6	42.1
$K = 64$	3.0		20.8	36.4	45.9	53.3	57.2	18.9	31.8	40.9	48.3	54.7	15.1	28.2	38.5	46.5	50.7	13.9	25.3	35.9	45.3	49.4
	5.0		27.9	41.1	49.8	57.3	60.9	24.6	37.3	46.2	53.9	59	19.7	33.4	43.9	51.3	55.9	16.8	31.1	41.3	50	53.6
	10.0		31.8	47.7	54.9	62	65.8	29.5	42.1	52.2	59.3	63.3	24.1	39.7	49.1	56	60.6	22.6	36.5	46.8	54.9	58.8
	NONE		50	66.1	73.3	77.2	79.1	48.9	62.5	71.5	76.3	78.4	45	62.2	70	74.9	78	45.2	60.9	70.1	75	77.7
	1.5		15.7	30.9	39.7	48.4	52.1	13.9	24.3	34.4	43.6	48.3	9.6	22.1	31.2	39.8	44.5	7.6	18.3	28.5	39	43.3
$K = 128$	3.0		26.2	40.7	49.5	57.1	59.9	21.7	34.7	43.7	51.3	56.7	16.9	30.4	40.4	48.1	52.1	15.1	27	37.3	46.8	50.8
	5.0		32.8	45.1	53.5	60.3	63.4	27.7	40.9	48.9	56.6	61.1	21.8	35.7	45.6	52.9	57.9	18.5	32.4	43.3	51.8	55
	10.0		36.2	51.8	59.4	65.5	68.2	33.1	45.4	54.6	61.8	65.9	26.2	41.7	52	58	62.4	24.2	38.2	49	56.5	60.5
	NONE		54.4	69.4	75.3	79.2	80.5	52.1	64.9	73.5	78.1	80	47.3	64.7	72.4	77.3	79.2	47.6	62.9	71	76.3	79
	1.5		21.4	37.7	45.3	51.2	53.4	19.3	28.4	37.5	46	50.7	11.9	24.4	33.5	42.4	47.1	8.9	19.9	29.7	39.8	44.3
$K = 256$	3.0		31.5	46.5	53.4	59.7	62	26.9	38.5	47	53.5	57.8	19.4	32.7	43	50.3	54.8	16.7	28.7	39	48.5	51.9
	5.0		38.6	50.6	57.1	64.4	65.2	32.2	43.8	51.7	59.4	63	24.2	38	48.3	55.2	59.1	20.3	34.4	44.3	52.5	56.1
	10.0		42.6	56.8	62.2	68.2	70.3	37.2	48.8	57.5	64.3	67.5	28.8	44.3	53.6	60.1	64.1	26.1	40.1	49.8	57.5	61.8

Table 6.2: Percentage of cut-edges for varying file popularity threshold (λ).

6.2.3 Partitioning Results based on the Ratio between Tasks and Files ($|T|/|F|$)

We fixed the zipf value as $\alpha = 1.0$ and the file popularity threshold as $\lambda = 1.5$, and we varied the ratio between tasks and files as 0.5, 1.0, 2.0, 5.0, 10.0 for each specific $|F|$. Figure 6.5 and Table 6.3 show the results of this experiment.

	$\|T\|/\|F\|$	$\|F\| = 1K$	$\|F\| = 2K$	$\|F\| = 5K$	$\|F\| = 10K$
	0.5	7.1	9.0	6.9	6.0
	1.0	23.2	19.4	18.4	16.0
$K = 32$	2.0	32.7	28.9	27.7	25.5
	5.0	43.0	39.1	36.6	35.5
	10.0	46.6	44.4	41.5	40.0
	0.5	10.0	10.7	8.1	6.6
	1.0	26.9	21.9	20.1	17.0
$K = 64$	2.0	36.3	31.7	29.4	27.0
	5.0	45.8	41.8	38.8	37.6
	10.0	49.7	46.8	43.4	42.1
	0.5	15.7	13.9	9.6	7.6
	1.0	30.9	24.3	22.1	18.3
$K = 128$	2.0	39.7	34.4	31.2	28.5
	5.0	48.4	43.6	39.8	39.0
	10.0	52.1	48.3	44.5	43.3
	0.5	21.4	19.3	11.9	8.9
	1.0	37.7	28.4	24.4	19.9
$K = 256$	2.0	45.3	37.5	33.5	29.7
	5.0	51.2	46.0	42.4	39.8
	10.0	53.4	50.7	47.1	44.3

Table 6.3: Percentage of cut-edges for varying $|T|/|F|$.

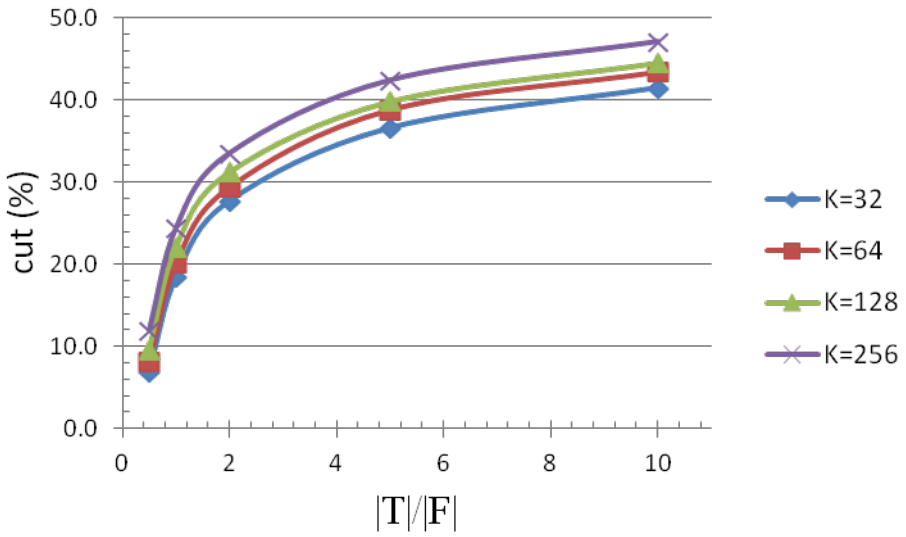

Figure 6.5: Cut (%) vs $|T|/|F|$ ($|F| = 1000,\ \lambda = 1.5,\ \alpha = 1.0$)

The results clearly show that in any case the edgecut is directly proportional to the ratio between tasks and files. This is because for a fixed number of files, when the ratio increases, files are requested more meaning that more edges for each file, so partitioning gets harder. Moreover, we realized that if we have more files along with a specific value of the ratio $|T|/|F|$, the corresponding edgecut values are smaller. In other words, if we have a larger graph with more tasks and files, the partitioning quality is higher. This is because once we partition a relatively smaller graph, we will end up with a low number of vertices per part whereas it is higher in larger graphs.

Also, the results given in Table 6.3 indicate that when we increase the number of sites (K) and the total number of files and tasks ($|T| + |F|$) with the same ratio, the cut percentages remain roughly same. For $K = 32$, consider $|T| = 2000$ and $|F| = 1000$ (so the ratio is as $|T|/|F| = 2$), and the cut percentage for these parameters is equal to 32.7%. When we check the cut ratio for $K = 64$, $|T| = 4000$ and $|F| = 2000$, it is equal to 31.7%, and we can see that is equal to first result. In addition to this, the following results for $|F| = 5000$, $|T| = 10000$ - $K = 128$ (assuming that it is nearly five times of $K = 32$) and $|F| = 10000$, $|T| = 20000$ - $K = 256$ (assuming that it is nearly ten times of $K = 32$) pairs are also close to the first result. Therefore, when we increase each parameter relatively, the cut

ratio remains same, as expected.

6.3 Improvement Results

Having partitioned the task-file graph described in Section 4.3, we map the parts to sites by using the algorithm in Section 5.2. In this section, we present the improvement results of these assignments accordingly. We have two specific parameters in Algorithm 4. Firstly, the τ parameter stands for the number of successive gains that leads to no improvement in the cost during the refinement. When we increase the value of τ, it is evident that we achieve better improvement values. This implies that tolerating a higher number of unprogressive gains leads us to come up with lower cost value. During the experiments, the τ parameter is set to 50. The second parameter is the number of initial mappings constructed randomly at the beginning. We computed the initial cost of each mapping and chose the one that provides the best cost. Throughout the experiments, we observed that increasing the number of initial randomly formed mappings led us to obtain better improvement results. We set this parameter to 50 throughout the experiments in this section.

The experiments fall under three main categories that are based on: the α values, the file popularity threshold (λ) and the ratio between tasks and files. The following sections discuss each of these categories and present the improvement results, respectively. Our improvement results take the initial random mapping as basis.

6.3.1 Improvement Results based on the Zipf Values

As was previously mentioned, we set three parameters to certain values as $\lambda = 1.5,\ |F| = 1000,\ |T| = 2000$ and varied zipfian threshold as $\alpha = 0.6,\ 0.8,\ 1.0,\ 1.2$. Figure 6.6 shows the improvement results based on varied Zipfian values. Also, Table 6.4 presents a more detailed view of these results.

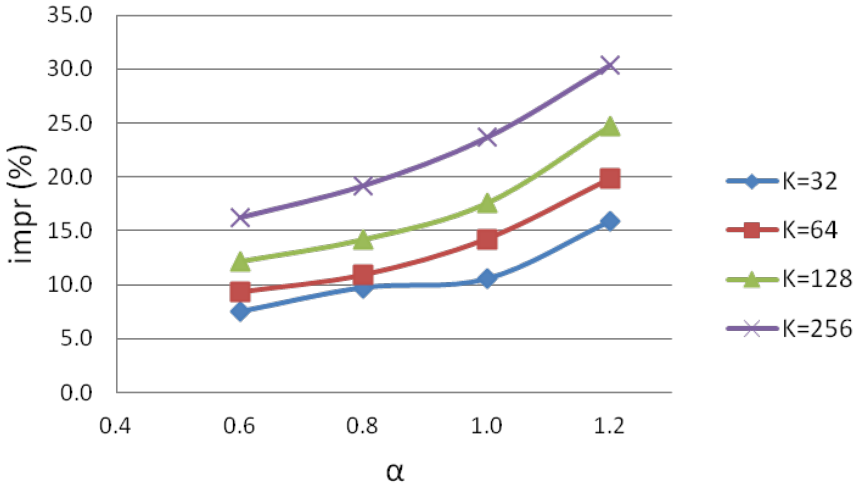

Figure 6.6: Improvement percentage for varying α values ($|F| = 1000$, $|T| = 2000$, $\lambda = 1.5$

The results indicate that the increase in the α values reflected positively on the improvement values. In Section 4.3, we firstly removed the file vertices that have high degrees. Since the determinant of the degrees of file vertices is Zipf distribution in our problem set, the increase in the zipf values makes more file vertices having higher degrees by diverging the distribution of files from uniformity. Hence, more files are removed by increasing zipf values, and this situation results in a decrease in the edgecut as explained in Section 6.2.1. The decrease in the edgecut means that less files will be transfered between sites, so this will improve the assignment between parts and sites.

		$\lvert F\rvert =$	$1K$					$2K$					$5K$					$10K$				
	zipf	$\lvert T\rvert =$	$0.5K$	$1K$	$2K$	$5K$	$10K$	$1K$	$2K$	$4K$	$10K$	$20K$	$2.5K$	$5K$	$10K$	$25K$	$50K$	$5K$	$10K$	$20K$	$50K$	$100K$
	0.6		17.3	10.5	7.5	6.0	4.0	12.6	8.2	6.4	4.8	3.1	8.8	5.8	4.9	4.0	2.9	7.4	4.8	4.2	3.4	2.6
$K = 32$	0.8		21.3	12.1	9.7	6.0	4.5	15.6	9.0	6.8	5.1	3.3	11.7	7.0	5.2	5.1	3.4	8.5	6.0	4.9	3.8	3.3
	1.0		33.7	15.4	10.5	7.6	5.6	22.0	13.4	8.7	6.1	4.9	15.0	9.5	6.6	5.2	4.3	13.9	7.8	5.9	4.2	3.8
	1.2		33.5	22.5	15.9	9.4	6.6	36.2	21.7	12.6	7.7	6.4	33.1	16.5	9.4	6.2	5.9	36.4	13.6	8.6	5.7	5.1
	0.6		22.7	13.5	9.3	5.8	4.1	17.3	9.9	7.5	4.7	3.4	11.8	7.2	5.0	3.9	2.7	8.9	5.5	4.1	3.5	2.5
$K = 64$	0.8		26.3	15.9	10.9	6.9	5.2	19.9	12.0	8.3	5.5	3.9	15.1	8.7	5.9	4.3	3.2	10.5	6.8	4.9	3.8	3.0
	1.0		37.0	20.2	14.2	9.0	6.5	28.0	17.2	10.9	7.0	5.3	21.3	11.2	8.0	5.3	4.5	17.5	9.6	6.3	4.2	3.7
	1.2		37.6	27.7	19.8	12.5	8.3	38.4	26.7	16.4	9.4	6.7	37.2	21.7	12.8	7.5	5.5	38.9	18.7	10.7	6.3	4.6
	0.6		27.4	17.5	12.1	7.7	5.4	21.6	13.5	9.0	5.5	3.8	15.2	9.0	6.0	3.9	2.7	11.3	6.4	4.5	3.1	2.4
$K = 128$	0.8		29.7	21.2	14.2	9.1	6.2	25.0	15.9	10.7	6.9	4.6	19.4	10.9	7.3	4.7	3.3	13.7	7.9	5.3	3.5	2.7
	1.0		38.2	25.8	17.6	11.2	8.1	33.0	22.5	14.5	8.9	6.3	27.4	15.1	10.1	6.3	4.5	22.3	12.5	7.6	4.8	3.6
	1.2		38.7	32.0	24.8	15.8	10.8	40.1	32.3	21.2	12.8	8.7	39.3	26.7	16.5	9.7	6.6	40.7	23.9	13.9	7.7	5.3
	0.6		33.2	23.9	16.2	10.2	7.2	27.5	18.0	12.4	7.5	5.1	20.7	12.6	8.3	5.0	3.5	15.2	9.1	5.9	3.7	2.7
$K = 256$	0.8		34.9	27.1	19.2	11.9	8.5	31.0	21.4	15.0	9.0	6.3	25.4	15.3	9.9	6.1	4.3	18.9	11.2	7.1	4.5	3.3
	1.0		40.5	31.3	23.6	14.8	11.0	36.7	28.2	19.5	12.3	8.4	32.8	20.3	13.7	8.3	5.9	27.9	16.6	10.8	6.1	4.3
	1.2		40.1	35.5	30.4	20.9	14.8	40.5	36.3	27.2	17.4	11.8	40.3	32.6	22.3	13.5	8.9	41.4	30.5	18.9	10.5	7.2

Table 6.4: Percentage of improvement for varying α values.

6.3.2 Improvement Results based on the File Popularity Threshold (λ)

We varied the λ as $\lambda = 1.5,\ 3.0,\ 5.0,\ 10.0$ and set the remaining parameters as $\alpha = 1.0$ and $|F| = 1000$. Table 6.5 shows the improvement values providing initial and final costs of mappings. Also, we illustrate the results in Figure 6.7.

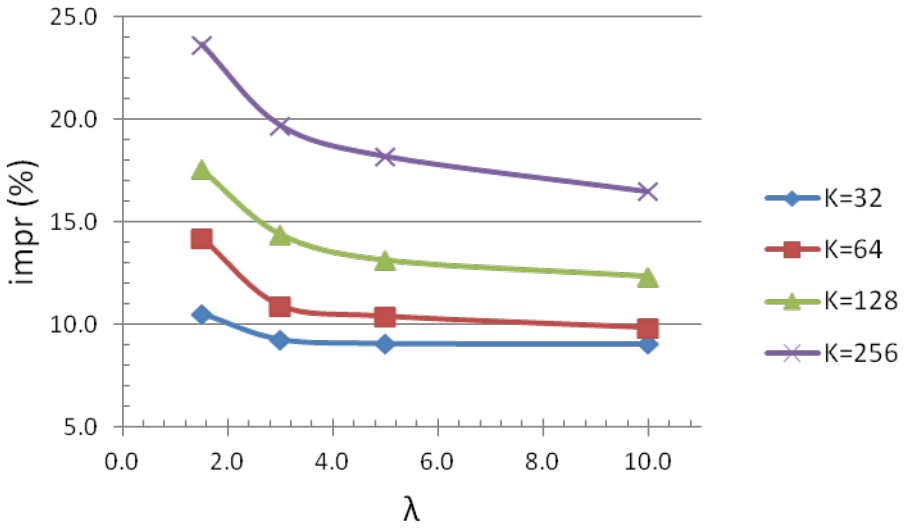

Figure 6.7: Improvement (%) vs λ ($|F| = 1000,\ |T| = 2000,\ \alpha = 1.0$)

The results show that the best improvement is achieved with minimum λ. The reason of this result is that the file popularity threshold is directly proportional to the edgecut between sites. Thus, the increase in the value of λ raises the edgecut that causes less improvement during mapping.

		$\lvert F\rvert =$	$1K$					$2K$					$5K$					$10K$				
	λ	$\lvert T\rvert =$	$0.5K$	$1K$	$2K$	$5K$	$10K$	$1K$	$2K$	$4K$	$10K$	$20K$	$2.5K$	$5K$	$10K$	$25K$	$50K$	$5K$	$10K$	$20K$	$50K$	$100K$
	1.5		33.7	15.4	10.5	7.6	5.6	22.0	13.4	8.7	6.1	4.9	16.0	9.5	6.6	5.2	4.3	13.9	7.8	5.9	4.2	3.8
$K = 32$	3.0		21.5	12.4	9.3	6.4	5.1	16.2	10.2	7.2	5.7	3.6	12.3	7.1	6.3	5.3	4.4	9.5	7.1	5.0	4.0	3.5
	5.0		17.0	12.7	9.1	5.8	4.6	15.1	8.8	6.5	5.7	4.0	9.1	7.2	5.4	4.4	3.0	9.2	6.5	5.2	4.2	3.9
	10.0		16.8	10.8	9.1	6.3	4.9	12.3	8.2	6.7	5.5	3.6	9.5	5.7	5.1	4.7	3.1	7.4	5.3	4.6	4.1	3.1
	1.5		37.0	20.2	14.2	9.0	6.5	28.0	17.2	11.1	7.6	5.3	21.3	11.2	8.0	5.3	4.5	17.5	9.6	6.3	4.2	3.7
$K = 64$	3.0		26.3	15.8	10.9	7.4	5.6	21.2	13.0	8.5	6.4	4.8	15.5	8.9	6.5	4.7	4.1	11.3	7.5	5.3	4.4	3.8
	5.0		22.9	14.1	10.4	6.9	5.4	17.9	11.5	7.9	5.8	4.7	12.6	8.0	6.1	4.6	3.1	10.8	6.6	5.3	4.1	3.5
	10.0		20.5	12.9	9.9	6.4	5.1	16.9	10.8	7.3	5.3	3.8	11.6	7.6	5.7	4.5	3.2	8.9	6.1	4.8	3.8	2.9
	1.5		38.2	25.8	17.6	11.2	8.1	33.0	22.5	14.5	8.9	5.9	27.4	15.1	10.1	6.3	4.5	22.3	12.5	7.6	4.8	3.6
$K = 128$	3.0		32.5	21.1	14.4	9.0	6.5	26.6	16.6	11.5	7.4	4.7	19.7	11.9	7.9	5.2	4.0	15.0	9.3	6.3	4.0	3.2
	5.0		28.7	19.1	13.1	8.3	6.1	23.6	14.8	10.1	6.6	4.8	17.1	10.4	7.0	4.9	3.4	13.3	7.8	5.5	3.8	3.0
	10.0		25.5	16.6	12.3	7.5	5.8	20.6	13.4	9.1	5.9	4.4	15.2	9.1	6.4	4.4	3.2	11.3	7.2	5.0	3.4	2.5
	1.5		40.5	31.3	23.6	14.8	11.0	36.7	28.2	19.9	12.3	8.4	32.8	20.3	8.4	8.3	5.9	27.9	16.6	10.8	6.1	4.3
$K = 256$	3.0		35.9	27.3	19.7	12.7	9.2	32.2	22.5	15.9	10.0	6.9	26.0	16.3	9.5	6.8	5.0	20.6	13.1	8.3	5.1	3.7
	5.0		33.1	25.1	18.2	11.6	8.6	28.4	20.0	14.1	8.9	6.3	22.3	15.1	10.2	6.4	4.5	18.2	11.2	7.4	4.6	3.5
	10.0		31.3	22.1	16.5	10.6	8.2	26.3	18.2	12.5	7.9	5.9	20.1	12.5	12.8	5.4	3.9	15.6	10.1	6.5	4.1	3.1

Table 6.5: Percentage improvement for varying file popularity threshold (λ).

6.3.3 Improvement Results based on the Ratio between Tasks and Files ($|T|/|F|$)

We varied the ratio as $|T|/|F| = 0.5,\ 1.0,\ 2.0,\ 5.0,\ 10.0$, and set the α value to 1.0 and λ to 1.5. The results are given in Figure 6.8 and Table 6.6.

	$\|T\|/\|F\|$	$\|F\| = 1K$	$\|F\| = 2K$	$\|F\| = 5K$	$\|F\| = 10K$
	0.5	33.7	22.0	15.0	13.9
	1.0	15.4	13.4	9.5	7.8
$K = 32$	2.0	10.5	8.7	6.6	5.9
	5.0	7.6	6.1	5.2	4.2
	10.0	5.6	4.9	4.3	3.8
	0.5	37.0	28.0	21.3	17.5
	1.0	20.2	17.2	11.2	9.6
$K = 64$	2.0	14.2	10.9	8.0	6.3
	5.0	9.0	7.0	5.3	4.2
	10.0	6.5	5.3	4.5	3.7
	0.5	38.2	33.0	27.4	22.3
	1.0	25.8	22.5	15.1	12.5
$K = 128$	2.0	17.6	14.5	10.1	7.6
	5.0	11.2	8.9	6.3	4.8
	10.0	8.1	6.3	4.5	3.6
	0.5	40.5	36.7	32.8	27.9
	1.0	31.3	28.2	20.3	16.6
$K = 256$	2.0	23.6	19.5	13.7	10.8
	5.0	14.8	12.3	8.3	6.1
	10.0	11.0	8.4	5.9	4.3

Table 6.6: Percentage of improvement for varying $|T|/|F|$.

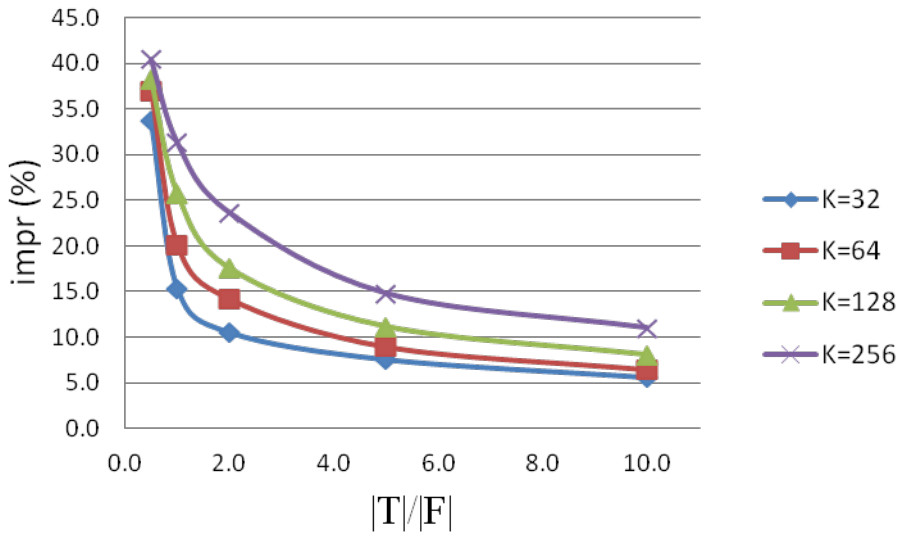

Figure 6.8: Improvement (%) vs $|T|/|F|$ ($|F| = 1000,\ \lambda = 1.5,\ \alpha = 1.0$)

As observed from the results, the improvement decreases with the increase in the ratio. When $|T|/|F|$ is smaller, the edges between files and tasks are sparse. As this ratio increases, the graph starts getting dense and partitioning dense graphs becomes difficult. Thus, after partitioning step, we get graphs that are poorer in quality in terms of edgecut as given in Section 6.2.3, and improving them becomes harder as seen in the results.

6.4 Replication Results

The first step of constructing task-file graphs is to create initial task-file graphs which are considered as the original ones. When we create these graphs, we only use the parameters that are the number of tasks ($|T|$), the number of files ($|F|$) and zipf values (α). For each and every possible value of these parameters within the boundaries we set, a task-file graph is constructed pertaining to all possible combinations of parameters. The total number of combinations is 80 where we have 5 alternatives for $|T|$, 4 alternatives for $|F|$ and 4 alternatives for α values as we have already presented in the previous experiments. Afterwards, for each existing task-file graph, we compute the average number of requests for each file

and specify a certain value that is the result of the file-request average multiplied by file popularity threshold (λ). Then we remove the files that have higher hits than this value from the graph. The removal operation is just done temporarily for the subsequent replication process.

Consequently, we now have two distinct task-file graphs: the initially-created original graphs and the trimmed graphs. We partition both of these graphs into K separate parts and replicate the removed files to all parts in the trimmed graphs. We experimented with original graphs and trimmed graphs with $\lambda = 1.5,\ 3.0,\ 5.0$ values while fixing $|T| = 4000$, $|F| = 2000$ and $\alpha = 1.0$. The partitioning results are given in Figure 6.9.

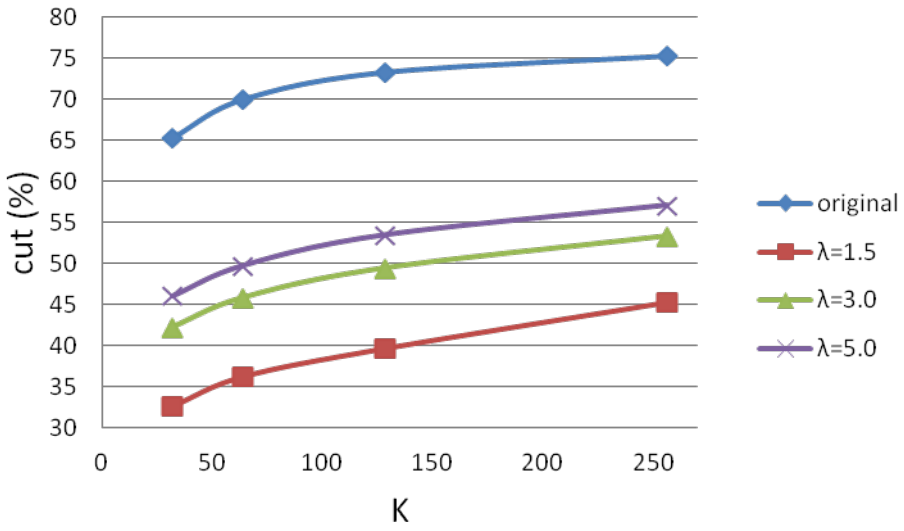

Figure 6.9: Cut percentage (%) of partitioning results for varying λ ($|F| = 2000, |T| = 4000, zipf = 1.0$)

We compared the percentage of cut edges of original graphs with the trimmed graphs for $\lambda = 1.5,\ 3.0,\ 5.0$. The results for $\lambda = 10.0$ are not given since these graphs are very similar to the original ones. We can make the inference from the results that the trimmed graphs are of better quality than the original ones considering related $cut(\%)$ values.

After the partitioning phase, as we described in Chapter 5, we have mapped the parts onto the sites and improved the mapping by swap operations. For each graph, we compute the communication cost which is the total file transfer load and try to improve this mapping. The effect of replication over the trimmed graphs has also been analyzed. Table 6.7 shows the improvement results along with the replication percentages and the final cost values. The improvement in the table is computed according to the difference between the cost of original graph and the decreased cost that file replication provides based on λ values. In the table, the results are presented by setting the following parameters as $\alpha = 1.0,\ K = 64$.

The results strongly suggest that replication decreases the communication cost and increases the improvement significantly. In addition, rather less improvement is obtained as the λ value increases which is because as λ increases, the replication ratios decrease. Moreover, once we keep the number of files fixed and increase the number of tasks, the obtained improvement is relatively lower due to the dense graph.

		Original	$\lambda = 1.5$			$\lambda = 3.0$			$\lambda = 5.0$			$\lambda = 10.0$		
	$\|T\|$	cost	repl (%)	cost	impr (%)	repl (%)	cost	impr (%)	repl (%)	cost	impr (%)	repl (%)	cost	impr (%)
$\|F\| = 1K$	$0.5K$	884	10.8	98	88.86	3.8	250	71.67	1.6	406	54.11	0.9	507	42.63
	$1K$	3079	9.3	505	83.60	4.2	894	70.97	2.6	1113	63.87	1.1	1562	49.28
	$2K$	7504	9.7	1295	82.74	4.2	2213	70.51	2.7	2639	64.84	1.4	3376	55.01
	$5K$	17950	9.2	4018	77.61	4.5	6138	65.81	2.7	7743	56.86	1.4	10040	44.07
	$10K$	38197	9.8	7997	79.06	4.5	12642	66.90	3.0	15217	60.16	1.3	20527	46.26
$\|F\| = 2K$	$1K$	2137	7.7	221	89.65	3.5	480	77.54	1.7	701	67.18	0.8	974	54.40
	$2K$	5473	9.5	747	86.36	3.8	1486	72.84	2.0	1919	64.94	1.0	2556	53.30
	$4K$	13755	8.9	2100	84.73	4.2	3584	73.95	2.5	4458	67.59	1.2	5883	57.23
	$10K$	32251	8.7	6841	78.79	4.4	10022	68.93	2.5	13067	59.48	1.2	16803	47.90
	$20K$	79017	8.8	14855	81.20	4.3	22392	71.66	2.5	27562	65.12	1.2	34535	56.29
$\|F\| = 5K$	$2.5K$	4563	6.5	448	90.18	2.8	992	78.25	1.5	1443	68.37	0.7	1919	57.94
	$5K$	14693	7.4	1869	87.28	3.6	3299	77.55	2.1	4432	69.84	0.9	6060	58.76
	$10K$	34876	7.8	5013	85.63	3.6	8320	76.14	2.1	10815	68.99	1.0	13936	60.04
	$25K$	89083	7.7	14762	83.43	3.8	22472	74.77	2.2	28700	67.78	1.1	36433	59.10
	$50K$	201346	7.7	32993	83.61	3.9	48412	75.96	2.2	61873	69.27	1.1	80040	60.25
$\|F\| = 10K$	$5K$	11724	6.3	757	93.54	2.6	1919	83.64	1.6	2543	78.31	0.7	3777	67.79
	$10K$	29910	7.0	3137	89.51	3.2	5712	80.90	1.8	7734	74.14	0.8	10447	65.07
	$20K$	64047	7.3	8701	86.41	3.4	14755	76.96	1.9	19194	70.03	0.9	24757	61.35
	$50K$	186458	7.1	28444	84.75	3.5	42522	77.19	2.0	54878	70.57	1.0	68808	63.10
	$100K$	362550	7.2	61374	83.07	3.5	91906	74.65	2.1	113406	68.72	1.0	145889	59.76

Table 6.7: Replication and improvement of communication cost (%) ($\alpha = 1.0,\ K = 64$)

Chapter 7

Conclusion

Data Grids provide geographically distributed resources for data intensive applications. To address the issues for better performance results, a combination of replication and scheduling strategies and a bipartite graph for modeling tasks and files in the *Data Grid* are proposed.

As previously mentioned, our grid system is composed of K sites which contain both storage and computing units. There are $|T|$ jobs and $|F|$ files in the system, and each job needs a subset of files as its input files. The relationship between files and jobs constructs the access log. This access log forms the basis of our approach which enables us to form a our bipartite graph model.

The first phase of our methodology is to partition the *task-file graph* into K parts. Firstly, we remove the most popular files within a certain threshold to be replicated later from the graph. In order to select these files, we first calculate average number of accesses to files. Subsequently, if a file is accessed more than the average multiplied by a threshold value λ, we remove this file to replicate it later to all parts. We have used METIS [29] to partition the constructed bipartite graphs. At the end of this step, we obtain a data placement and job scheduling strategy. In the next phase, we map the obtained parts to the sites using a *KL-based heuristic*. At this phase, we initially assign the parts to the sites in a random fashion and calculate their initial costs. At each and every step of

refinement, we try to decrease the cost, and when no more improvement can be obtained, we take this mapping and use it for mapping tasks and files to sites. After data placement and job scheduling, the next phase is data replication. The most accessed files have been identified before the partitioning and mapping step, and these files are replicated to all sites.

Ee can summarize the steps under our methodology as follows:

(i) Access logs are constructed.

- ◇ The parameters to be varied are number of jobs ($|T|$) and number of files ($|F|$).
- ◇ Zipf distribution is used to simulate file popularity.
- ◇ Uniform distribution is used to determine the number of files that each task requests.

(ii) Data grid environment is constructed.

- ◇ Various environments are created according to different number of sites ($|K|$).

(iii) Task-file graph is partitioned into K parts.

(iv) Parts are mapped onto the sites.

- ◇ KL-based refinement algorithm.

(v) Replication

- ◇ The popular files in the access logs are replicated to all sites.

Experiments are analyzed in three categories as partitioning results, improvement results and replication results. The experiments indicate that we can take advantage of access logs to exploit relations among tasks and files, and use this to reduce total transfer cost. Furthermore, by using a simple replication strategy and task-file placement methodology, we can reduce total communication cost significantly.

Bibliography

[1] CERN - The European Organization for Nuclear Research, `http://public.web.cern.ch/public/`. Accessed August 25, 2012.

[2] D. Nukarapu, B. Tang, L. Wang, and S. Lu, "Data replication in data intensive scientific applications with performance guarantee," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, pp. 1299–1306, Aug. 2011.

[3] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data management in an international data grid project," in *Grid Computing GRID 2000* (R. Buyya and M. Baker, eds.), vol. 1971 of *Lecture Notes in Computer Science*, pp. 333–361, Springer Berlin / Heidelberg, 2000.

[4] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman, "A metadata catalog service for data intensive applications," in *Supercomputing, 2003 ACM/IEEE Conference*, p. 33, nov. 2003.

[5] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pp. 352–, IEEE Computer Society, 2002.

[6] E. Laure, H. Stockinger, and K. Stockinger, "Performance engineering in data grids: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, pp. 171–191, Feb. 2005.

[7] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management

and transfer in high-performance computational grid environments," *Parallel Comput.*, vol. 28, pp. 749–771, May 2002.

[8] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23, no. 3, pp. 187 – 200, 2000.

[9] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman, "Data replication strategies in grid environments," in *in Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP02*, pp. 378–383, Press, 2002.

[10] P. Avery and I. Foster, "The GriPhyN Project: Towards Petascale Virtual-Data Grids," Tech. Rep. GriPhyN 2001-14, The GriPhyN Collaboration, 2001.

[11] H. Lamehamedi, Z. Shentu, B. Szymanski, and E. Deelman, "Simulation of dynamic data replication strategies in data grids," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, p. 10 pp., april 2003.

[12] P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger, "File-based replica management," *Future Generation Computer Systems*, vol. 21, no. 1, pp. 115 – 123, 2005.

[13] U. Čibej, B. Slivnik, and B. Robič, "The complexity of static data replication in data grids," *Parallel Comput.*, vol. 31, pp. 900–912, Aug. 2005.

[14] M. Tang, B.-S. Lee, C.-K. Yeo, and X. Tang, "Dynamic replication algorithms for the multi-tier data grid," *Future Gener. Comput. Syst.*, vol. 21, pp. 775–790, May 2005.

[15] R.-S. Chang and H.-P. Chang, "A dynamic data replication strategy using access-weights in data grids," *J. Supercomput.*, vol. 45, pp. 277–295, Sept. 2008.

[16] K. Ranganathan and I. Foster, "Design and evaluation of dynamic replication strategies for a high-performance data grid," in *International Conference on Computing in High Energy and Nuclear Physics*, 2001.

[17] K. Ranganathan, A. Iamnitchi, and I. Foster, "Improving data availability through dynamic model-driven replication in large peer-to-peer communities," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pp. 376–, 2002.

[18] R. M. Rahman, K. Barker, and R. Alhajj, "Replica placement in data grid: Considering utility and risk," in *Proceedings of the International Conference on Information Technology: Coding and Computing*, ITCC '05, pp. 354–359, 2005.

[19] S. min Park, J. hoon Kim, Y. bae Ko, and W. sik Yoon, "Dynamic data grid replication strategy based on internet hierarchy," in *In Second International Workshop on Grid and Cooperative Computing (GCC2003)*, 2003.

[20] M. Lei, S. V. Vrbsky, and X. Hong, "An on-line replication strategy to increase availability in data grids," *Future Gener. Comput. Syst.*, vol. 24, pp. 85–98, Feb. 2008.

[21] Y.-H. Lee, S. Leu, and R.-S. Chang, "Improving job scheduling algorithms in a grid environment," *Future Gener. Comput. Syst.*, vol. 27, pp. 991–998, Oct. 2011.

[22] K. S. Chatrapati, J. U. Rekha, and A. V. Babu, "Recursive competitive equilibrium approach for dynamic load balancing a distributed system," in *Proceedings of the 7th international conference on Distributed computing and internet technology*, ICDCIT'11, pp. 162–174, Springer-Verlag, 2011.

[23] A. Silberschatz, P. Galvin, and G. Gagne, "Operating system concepts, sixth edition," 2002.

[24] T. Rings, G. Caryer, J. R. Gallop, J. Grabowski, T. Kovacikova, S. Schulz, and I. Stokes-Rees, "Grid and cloud computing: Opportunities for integration with the next generation network," *J. Grid Comput.*, vol. 7, no. 3, pp. 375–393, 2009.

[25] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–131, 1999.

[26] M. Tang, B.-S. Lee, X. Tang, and C.-K. Yeo, "Combining data replication algorithms and job scheduling heuristics in the data grid," in *Euro-Par 2005 Parallel Processing* (J. Cunha and P. Medeiros, eds.), vol. 3648 of *Lecture Notes in Computer Science*, pp. 614–614, Springer Berlin / Heidelberg, 2005.

[27] B. Hendrickson and R. Leland, "The Chaco User's Guide: Version 2.0," Tech. Rep. SAND94–2692, Sandia National Lab, 1994.

[28] C. Walshaw and M. Cross, "Jostle: parallel multilevel graph-partitioning software an overview," tech. rep., Computing and Mathematical Sciences, University of Greenwich, Old Royal Naval College, Greenwich, London.

[29] G. Karypis and V. Kumar, *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[30] B. Monien and S. Schamberger, "Graph partitioning with the party library: Helpful-sets in practice," in *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '04, pp. 198–205, IEEE Computer Society, 2004.

[31] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, pp. 318–331, July 2008.

[32] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell system technical journal*, vol. 49, no. 1, pp. 291–307, 1970.

[33] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation, 1982. 19th Conference on*, pp. 175 –181, june 1982.

[34] K. Ranganathan and I. T. Foster, "Identifying dynamic replication strategies for a high-performance data grid," in *Proceedings of the Second International Workshop on Grid Computing*, GRID '01, pp. 75–86, Springer-Verlag, 2001.

[35] M. Carman, F. Zini, L. Serafini, and K. Stockinger, "Towards an economy-based optimisation of file access and replication on a data grid," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pp. 340–, 2002.

[36] W. H. Bell, D. G. Cameron, R. Carvajal-Schiaffino, A. P. Millar, K. Stockinger, and F. Zini, "Evaluation of an economy-based file replication strategy for a data grid," in *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, p. 661, 2003.

[37] G. Khanna, N. Vydyanathan, T. Kurc, U. Catalyurek, P. Wyckoff, J. Saltz, and P. Sadayappan, "A hypergraph partitioning based approach for scheduling of tasks with batch-shared i/o," in *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '05, pp. 792–799, IEEE Computer Society, 2005.

[38] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pp. 30–, IEEE Computer Society, 1999.

[39] M. Tang, B.-S. Lee, X. Tang, and C.-K. Yeo, "The impact of data replication on job scheduling performance in the data grid," *Future Gener. Comput. Syst.*, vol. 22, pp. 254–268, Feb. 2006.

[40] G. Khanna, N. Vydyanathan, U. Catalyurek, T. Kurc, S. Krishnamoorthy, P. Sadayappan, and J. Saltz, "Task scheduling and file replication for data-intensive jobs with batch-shared i/o," in *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pp. 241 –252, 2006.

[41] A. Chakrabarti, R. A. Dheepak, and S. Sengupta, "Integration of scheduling and replication in data grids," pp. 375–385, 2005.

[42] A. Elghirani, R. Subrata, and A. Y. Zomaya, "Intelligent scheduling and replication in datagrids: a synergistic approach," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pp. 179–182, IEEE Computer Society, 2007.

[43] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *In INFOCOM*, pp. 126–134, 1999.

[44] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Addison-Wesley (Reading MA), 1949.

[45] R. E. Burkard, E. Cela, P. M. Pardalos, and L. S. Pitsoulis, "The quadratic assignment problem," 1998.