A Data-Locality Aware Mapping and Scheduling Framework for Data-Intensive Computing

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Gaurav Khanna, B.E. (Hons.), M.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:

Prof. P. Sadayappan, Adviser

Prof. Joel H. Saltz

Prof. Umit V. Catalyurek

Prof. Tahsin M. Kurc

Prof. Srinivasan Parthasarathy

Approved by

Adviser

Graduate Program in Computer Science and Engineering

ABSTRACT

Science is increasingly becoming more and more data-driven. With technological advancements such as advanced sensing technologies that can rapidly capture data at high resolutions and Grid technologies that enable increasingly realistic simulation of complex numerical models, scientific applications have become very data-intensive and involve storing and accessing large amounts of data. The LHC experiment at CERN is an example of a high energy physics initiative where the amount of data stored is in petabytes. The end goal in collecting petabytes of simulation data is to gain a better understanding of the problem under study. This essentially involves collaborative analysis of data by scientists across the world which conforms to a distributed data-intensive computing paradigm where a set of compute, storage and network resources are used in a collective fashion to advance science. Effective scheduling and resource management for such data-intensive applications on distributed resources is critical in order to meet their performance requirements.

Efficient scheduling in the aforementioned scenario encompasses two key interrelated problems. The first one is the data staging problem which involves the staging of data from the simulation/experimental sites to the computational sites where the data analysis needs to be performed. The second one is the job mapping problem which involves the mapping of data analysis jobs to compute resources in such a manner so as to maximize the locality of data usage.

Traditional batch job schedulers are designed for compute-intensive jobs running at supercomputer centers. They take into account CPU related metrics (e.g., user estimated job run times) and system state (e.g., queue wait times) to make scheduling decisions, but they do not take into account data related metrics. Therefore, there is a need for designing scheduling mechanisms for data-analysis jobs that take into account not only the computation time of the jobs, but also the overheads of retrieving files requested by those jobs.

In this dissertation, we address the problem of data staging and job mapping for data-intensive jobs in both homogeneous and heterogeneous environments. We achieve this by taking into account the effects of data staging, end-point contention and data locality. For the mapping problem, we propose algorithms for mapping data-intensive jobs in both an offline and online setting. We also study the interplay between job mapping and data replication and propose algorithms which perform job mapping and data replication in a coordinated manner. For the data staging problem, we propose efficient data staging mechanisms for data centers consisting of coupled collections of storage and compute clusters. Furthermore, we extend our data staging work to a heterogeneous distributed system like the Grid. To accomplish that, we employ multi-hop path splitting and multi-pathing optimizations to improve wide-area file transfer throughput.

To my parents, Shr. Ashok Khanna and Smt. Anu Khanna

ACKNOWLEDGMENTS

First of all, I would like to acknowledge the support of my adviser, Prof. P. Sadayappan and my co-advisers, Prof. Umit Catalyurek, Prof. Tahsin Kurc and Prof. Joel Saltz for the invaluable support and guidance that they have given me over the past 4 years of my graduate study. I am especially grateful to Prof. Saday. He has always been available for whatever advice I needed, be it academic/research related or other matters. His visionary thoughts, perspective on research and energetic working style have influenced me greatly as a researcher. I am extremely fortunate to have had the opportunity to work with and learn from him during the past several years. I could not have hoped for a better adviser. I would like to thank Umit and Tahsin for being deeply involved with my research and always willing to help me with new ideas and directions. I would also like to thank Prof. Srini Parthasarthy for his valuable guidance and suggestions.

I am very thankful to my colleague and fellow labmate Nagavijayala Vydyanathan for the innumerable discussions and collaborations relating to this research. Her patience and dedication towards work has motivated me a lot especially during the tough times I had during the early days of my graduate research. I would like to thank my labmates, Qingda Lu, Albert Hartono and Uday Bondhugula. They were all ever willing to be of help on research related questions and we also had many lively discussions on general topics.

Graduate school has been a roller coaster for me. I, like every other Phd. student, have to had to deal with qualifying exam, the candidacy exam, the many tight paper deadlines, paper rejections and above all phases of stagnation in research. I would like to thank my close friends (in alphabetical order), Abhilash, Muthu, Raj, Sitaram, Vijay for their constant support and encouragement through my good and bad times. Thanks to Raj for being an excellent roommate for 3 years. Thanks to Vijay, for we had lot of common things to share as far as our research was concerned. Thanks to Sitaram for always patiently listening to my random thoughts. I thank them, for together we have enjoyed endless dinners, movies, cricket, gym sessions, mindless discussions and what not. I would also like to thank Amol and Vishnu for many discussions on research/job related matters and Amol again for being always there to organize our cricket games. Also, thank Ambrish for those coffee breaks and memories of Delhi and DIT.

Finally, I would want to thank the two most important people in my life, my dad Shr. Ashok Khanna and my mother Smt. Anu Khanna. I feel a deep sense of gratitude towards them for being pillars of support and encouragement in all my endeavors. Without their support, nothing would have been possible. I owe them everything. I am greatly indebted to my younger brother, Nitesh, and both my grandmothers, Smt. Indira Khanna and Smt. Nirmal Malhotra for their continued love and affection towards me.

VITA

April 13, 1981	Born - Ranchi, India
2002	B.E. (Hons.) Computer Engineering Delhi University of Technology, University of Delhi, India
June 2006 - Sep. 2006	Research Intern, IBM India Research Lab.
June 2007 - Sep. 2007	Research Intern, MCS Division, Argonne National Lab.
2003-2004	Graduate Teaching Associate, The Ohio State University.
2004-2008	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Gaurav Khanna, Umit V. Catalyurek, Tahsin M. Kurc, Raj Kettimuthu, Ponnuswamy Sadayappan, Ian Foster and Joel H. Saltz "Multi-Hop Path Splitting and Multi-Pathing Optimizations for Data Transfers over Shared Wide-Area Networks using GridFTP (Short Paper)". *To Appear in the Proceedings of the 17th IEEE International Symposium on High-Performance Distributed Computing*, 2008.

Gaurav Khanna, Umit V. Catalyurek, Tahsin M. Kurc, Raj Kettimuthu, Ponnuswamy Sadayappan and Joel H. Saltz "A Dynamic Scheduling Approach for Coordinated Wide-Area Data Transfers using GridFTP". *In Proceedings of the 22th IEEE International Parallel and Distributed Processing Symposium*, 2008.

Gaurav Khanna, Umit V. Catalyurek, Tahsin M. Kurc, Ponnuswamy Sadayappan and Joel H. Saltz "Scheduling File Transfers for Data-Intensive Jobs on Heterogeneous

Clusters". *In Proceedings of the 13th European Conference on Parallel and Distributed Computing*, 214–223, 2007.

John Bresnahan, Mike Link, Gaurav Khanna, Zulfikhar Imani, Rajkumar Kettimuthu, Ian Foster "Globus GridFTP: What's New in 2007 (Invited Paper)". *In Proceedings of the 1st International Conference on Networks for Grid Applications*, 2007.

Gaurav Khanna, Nagavijayalakshmi Vydyanathan, Umit V. Catalyurek, Tahsin M. Kurc, Sriram Krishnamoorthy, Ponnuswamy Sadayappan, and Joel H. Saltz "Task Scheduling and File Replication for Data-intensive Jobs with Batch-shared I/O". *In Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 241–252, 2006.

Gaurav Khanna, Umit V. Catalyurek, Tahsin M. Kurc, Ponnuswamy Sadayappan and Joel H. Saltz "A Data Locality Aware Online Scheduling Approach for I/O-Intensive Jobs with File sharing". *In Proceedings of the 12th Workshop on Job Scheduling Strategies for Parallel Processing*, 141–160, 2006.

Nagavijayalakshmi Vydyanathan, Gaurav Khanna, Umit V. Catalyurek, Tahsin M. Kurc, Ponnuswamy Sadayappan, and Joel H. Saltz "Scheduling of Tasks with Batch-shared I/O on Heterogeneous Systems". *In Proceedings of the Heterogeneous Computing Workshop*, 2006.

Gaurav Khanna, Nagavijayalakshmi Vydyanathan, Tahsin M. Kurc, Umit V. Catalyurek, Pete Wyckoff, Joel H. Saltz, and Ponnuswamy Sadayappan "A Hypergraph Partitioning Based Approach for Scheduling of Tasks with Batch-shared I/O". *In Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, 792–799, 2005.

Nagavijayalakshmi Vydyanathan, Gaurav Khanna, Tahsin M. Kurc, Umit V. Catalyurek, Pete Wyckoff, Joel H. Saltz, and Ponnuswamy Sadayappan "Use of PVFS for Efficient Execution of Jobs with Pipeline-shared I/O". *In Proceedings of the IEEE/ACM International Workshop on Grid Computing*, 235–242, 2004.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

List of Algorithms

CHAPTER 1

INTRODUCTION

Science is increasingly becoming more and more data-driven. The ability of a geographically distributed community of scientists to access and analyze large amounts of data has emerged as a significant requirement for furthering science. Such a requirement occurs both in simulation science applications like climate modeling and experimental science applications like high energy physics. In simulation science, the data is generated via supercomputer simulations. For example, the DOE-funded climate modeling project [19] involves accessing and processing of climate simulation data. Similarly, experimental science also involves data analysis where the data sources are scientific facilities like high-energy physics experiments.

The common aspect of these simulations and experimental endeavors is the large scale of data which currently ranges in petabytes. This is facilitated by the advent of advanced sensing technologies that can rapidly capture data at high resolutions and Grid technologies that enable increasingly realistic simulation of complex numerical models. The DZero [3] experiment at the Department of Energy's (DOE) Fermi National Accelerator Laboratory, BaBar [2] experiment at the Stanford Linear Accelerator Center (SLAC) and the LHC [5] experiment at CERN are some of the high energy physics initiatives where the amount of data stored is in petabytes.

Raw simulation data generated at a supercomputer center or experimental data generated through high-energy physics experiments is just a starting point for investigation. The end goal is to gain a better understanding of the problem under study. This essentially involves analysis, visualization and exploration of data thereby enabling a better understanding of the problem under study and a more efficient refinement of the search space of solutions. Data analysis involves accessing and processing of many subsets of a dataset. Most scientific datasets are stored in files. These files typically reside on storage archives which may be centralized or distributed across multiple individual storage sites. The storage archives are typically placed closed to the simulation sites or the experimental facilities. While for simulation data, the data analysis can be performed at the simulation site itself, very often analysis needs to be performed on a different set of resources. In other words, a scientist performs data analysis by downloading regions of interest in the dataset to local computational sites where the data analysis needs to be performed. These computational sites could be scientists' local sites or a shared computational resource like a cluster. A region of interest specifies a subset of data files and/or segments in data files either directly as input parameters or after an index lookup that finds the files and file segments of interest. Simulations are then executed on "local" computational resources to produce additional scientific data.

In order to produce useful scientific results, data analysis is performed by scientists around the world. Such a system model conforms to a distributed data-intensive computing paradigm where a set of compute, storage and network resources are used in a collective fashion to advance science. Efficient job scheduling in such a distributed heterogeneous multi-site scenario is a key problem since the transfer of data from the

simulation sites to the scientists local sites is a bottleneck. Moreover, the resources are heterogeneous and distributed. The existence of multiple replicas of each dataset on locations with different networking and storage capabilities adds an added dimension to the range of choices for appropriate selection of resources for a job followed by scheduling of the job on the selected resources.

1.1 Application Scheduling

Efficient scheduling in the aforementioned scenario encompasses two key inter-related problems. One of them is the data staging problem which involves the staging of data from the simulation sites to the computational sites where the data analysis needs to be performed. This will involve coordination of data movement across multiple source sites, destination sites and intermediate locations, and among multiple users and applications. The other one is the job mapping problem which involves the mapping of data analysis jobs to compute resources in such a manner so as to minimize the completion time of the jobs. This will involve mapping jobs to compute sites in such a way so as to minimize the data staging cost while at the same time attaining load balance across different computing sites.

1.1.1 Mapping/Scheduling of Jobs

Scheduling of jobs has been active area of computer science research. Most of the existing batch schedulers like the Portable Batch System (PBS) [40], Load Sharing Facility (LSF) [77] etc. have been proposed in the context of compute-intensive jobs running at supercomputer centers. They take into account CPU related metrics (e.g., user estimated job run times) and system state (e.g., queue wait times) to make scheduling decisions, but they do not take into account data related metrics.

Therefore, there is a need for designing scheduling mechanisms for data-analysis jobs take into account not just the availability of computation sites, but also the storage sites and the networking resources connecting the storage sites to the computational sites. We believe that addressing the following set of issues will enable us address the data-intensive job scheduling problem to a reasonable extent.

1. **Storage Affinity:** A computational site is typically a homogeneous computational cluster which consists of a set of processing nodes. Each node of the computational cluster has a local disk, where files from storage sites can be transferred and "staged" prior to the execution of the executable. Therefore, each newly arriving data analysis job has an affinity with each node of the computational cluster nodes in that the files requested by each job may already be present on some of the local nodes due to prior job executions. In order to make the scheduler "data-aware", the file locality information needs to be considered when making scheduling decisions. Moreover, jobs have varying execution times. Therefore, in addition to considering data affinity, job allocations should be done in such a manner to ensure computational load balance among the processing nodes.

2. **Data Sharing/Reuse:** With large data files, it is especially important to carefully consider data reuse. Due to limited storage space on the compute site, staging new files may require existing files to be evicted. The key issue is to employ an efficient caching/replacement policy which can keep popular files for a longer period of time while evicting less popular files when evictions are required. An example scenario of data-reuse arises when multiple scientists located at geographically different physical sites issue data staging requests which

contain an overlap of requested data. In this instance it can be possible for one of the users to retrieve the information from the other users data repository, rather then the remote storage server. Another scenario pertains to parameter sweep searches where the scientist is trying to extract knowledge from a dataset but does not know which subset of the data will provide the required knowledge. In this case, the scientist sends multiple data-analysis queries which have overlapping portions. It would be beneficial if the necessary files are cached locally rather then retrieved from global storage for each job's execution. These scenarios emphasize that data is reused among multiple analysis queries and this fact can be used to make better file caching/replacement decisions.

3. **Replication of Popular Datasets:** The existence of multiple distributed storage sites entails incorporating a data replication policy which keeps multiple copies of important files in order to minimize the contention in accessing such hot-spots by multiple data requests. Executing jobs at compute sites, leads to the creation of new copies of the requested files, if not already present at the requesting location. On the other hand, replication of files to multiple compute sites drives scheduling decisions, which are trying to exploit file affinity cause by existence of the replica. Scheduling of jobs and the replication of files, therefore, share a symbiotic relationship that needs to be accounted for to ensure efficient execution of jobs.

1.1.2 Data Staging

Efficient transfer of data from the simulation sites to the scientists' local sites is a key problem since the data analysis is often performed on a cluster or a scientist's

desktop and not the simulation sites where the data is generated. An example scenario for data staging is the LHC [5] experiment at CERN. The data which is generated by a CMS experiment at LHC needs to be transferred to the Tier-1 site in the US where it is processed and then multicast onto many domestic US tier-2 sites. Staging of data from the data source to the sinks involves the following key issues.

1. Movement of data over wide-area networks encompassing a diversity of sources and sinks.

2. Coordinated data movement which takes into account network, storage access bandwidths at each resource. In addition, the end-point contention at each resource needs to be modeled in order to avoid load-imbalance.

3. Data replication mechanisms should be integrated with data delivery systems in order to store more popular data for the interests of the scientific community.

4. Need to incorporate a level of predictability in the data transfer times in spite of network load fluctuations.

1.2 Contributions

In this dissertation, we investigate the aforementioned job scheduling and data staging problem by taking into account the effects of data staging, end-point contention and data locality. Specifically, our contributions include the following.

1. **Job Mapping in both Offline and Online context**

   In Chapter 3, we present job mapping algorithms for a batch of data-analysis jobs on coupled compute and storage platforms. The platform consist of storage

nodes which are connected to a pool of compute nodes (compute cluster) over a local area network. Each compute node has one or more local disks and can request files from any of the storage nodes. Such configurations are likely to be common in institutions as well as supercomputer centers, since compute and storage clusters are designed with different goals in mind. We achieve data locality by modeling the job-file sharing patterns using a hypergraph and employing hypergraph partitioning to get a load-balanced cut-minimized partitioning of jobs onto compute nodes [51], [49].

2. **Coordinated Job Scheduling and Data Replication**

   Chapter 4 of this dissertation studies the interplay between job mapping and data replication and proposes algorithms which perform job mapping and data replication in a coordinated manner. In this work, we also account for the limited storage space constraints on each node of a compute cluster. We propose a 0-1 Integer programming formulation of the aforesaid problem. In addition, we also propose hypergraph-partitioning based heuristics to solve this problem [50].

3. **Data Staging**

   In Chapter 5 , we propose efficient ways of scheduling file transfers from a set of source nodes to a set of destination nodes in the context of data centers. Data centers consisting of collections of storage and compute clusters provide a viable environment for hosting large scientific datasets and providing analysis services. We formulate the data staging problem using 0-1 Integer programming by employing concepts from network flows and time-expanded networks. This approach enables us to perform global optimization thereby obtaining better

schedules. However, the scheduling time of such an approach is very high which makes it unsuitable to be used for large scale systems and workload configurations. To address this issue, we propose a fast graph matching based heuristic approach which tries to maximize the parallelism and minimize the contention while scheduling the file transfers [46].

Chapter 6 outlines efficient algorithms to schedule and execute the transfer of a set of files distributed across multiple machines to another set of machines in a wide-area environment. We present a network flow based mixed integer programming (IP) formulation of the scheduling problem. The resulting solution is a lower bound on transfer time under idealistic conditions of resource availability and performance. We then propose a dynamic scheduling heuristic which employs network bandwidth information obtained from past GridFTP transfers to adapt its scheduling decisions, thereby, accounting for the resource availability fluctuations in the wide-area environment. The algorithm also employs adaptive replica selection, if files are replicated in the environment during previous transfers. It performs simultaneous transfer of portions of files from multiple replicas to maximize data transfer bandwidth. We have developed an implementation of our algorithm using GridFTP [11] as the underlying transport protocol for data transfers [48].

In Chapter 7, we explore the effects of multi-hop path splitting and multi-pathing to improve the file transfer performance in GridFTP. Multi-hop path splitting improves performance by replacing a direct TCP connection between the source and destination by a multi-hop chain through some intermediate nodes. Multi-pathing involves striping the data at the source and sending it

across multiple overlay paths thereby leading to a better achievable throughput. In other words, multiple independent routes can be employed to simultaneously transfer disjoint chunks of a file to its destination. We propose a path determination heuristic which incorporates these optimizations for efficient transfer of a single file. To optimize performance for batch file transfer requests, we extend the collective file-transfer scheduling heuristic discussed in Chapter 6. The extended algorithm incorporates multi-hop path splitting and multi-pathing optimizations.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we provide the background information of our work. First, we highlight the motivating applications for our work followed by a discussion on a typical data-intensive computing environment. This is followed by a discussion about job scheduling. We discuss the state-of-the-art in compute-intensive job scheduling and highlight the inadequacies of the current compute job scheduling systems when applied to the data-intensive context. This is followed by a discussion of some of the current literature in the data-intensive scheduling domain. Finally, we talk about the prior work which has been done to address the data-staging problem.

2.1 Data-Intensive Applications

Several scientific applications store datasets in collections of files. A request for data analysis specifies a subset of data files, either as a parameter of the request or through an index lookup that locates the files (or file segments) that satisfy the request. The data of interest is retrieved from the storage system and transformed into a data product, which is more suitable for examination by the scientist.

2.1.1 High-energy physics

High energy physics (HEP) is intended at making breakthroughs in the understanding of the fundamental interactions and structures that govern the nature of matter and space-time in our universe. HEP experiments are often based on finding one or several types of rare events. These rare events are typically generated by collision of particles. However, due to the rareness of interesting events, millions of collisions may be required to generate a few useful events. Such a process involves the generation of huge amounts of simulation data. Data analysis is performed simultaneously by a large collaboration of physicists usually working from their local centers distributed world-wide. The DZero [3] experiment at the Department of Energy's (DOE) Fermi National Accelerator Laboratory, BaBar [2] experiment at the Stanford Linear Accelerator Center (SLAC) and the LHC [5] experiment at CERN are some of the high energy physics projects involving petabytes of data.

2.1.2 Biomedical Image analysis

Biomedical imaging is a powerful method for disease (e.g., cancer) diagnosis and for monitoring therapy. State-of-the-art studies make use of large datasets, which consist of time dependent sequences of 2D and 3D images from multiple imaging sessions. Systematic development and assessment of image analysis techniques requires an ability to efficiently invoke candidate image quantification methods on large collections of image data. A researcher may apply several different image analysis methods on image datasets containing thousands of 2D and 3D images to assess ability to predict outcome or effectiveness of a treatment across patient groups.

2.1.3 Satellite Data processing

Remotely sensed data is either continuously acquired or captured on-demand via sensors attached to satellites orbiting the earth [27]. Datasets of remotely sensed data can be organized into multiple files. Each file contains a subset of data elements acquired within a time period and a region of the earth surface. For instance, a dataset in the form of a snapshot of the surface captured by a Landsat thematic mapper satellite consists of N files (usually 4 or 5 files), with each file corresponding to a specific sensor on the satellite and storing data captured by the sensor within the time period and surface region specified by the ground control. When multiple scientists access these datasets, there will likely be overlaps among the set of files requested because of "hot spots" such as a particular region or time period that scientists may want to study.

2.2 Typical Data-Intensive Computing Environment

A data-intensive computing environment consists of scientific applications (e.g., High Energy Physics (HEP), climate modeling, biomedical imaging) which involve data-analysis jobs that analyze the huge volumes of data generated via emerging experimentations and simulations. These datasets typically reside on storage archives which may be centralized or distributed across multiple individual storage sites. The datasets are accessed by scientists in geographically different locations who perform data analysis by either downloading the datasets locally or creating replicas of datasets to reduce the latencies involved in wide-area data transfers, thereby improving application performance. Simulations are then executed on "local" resources to produce

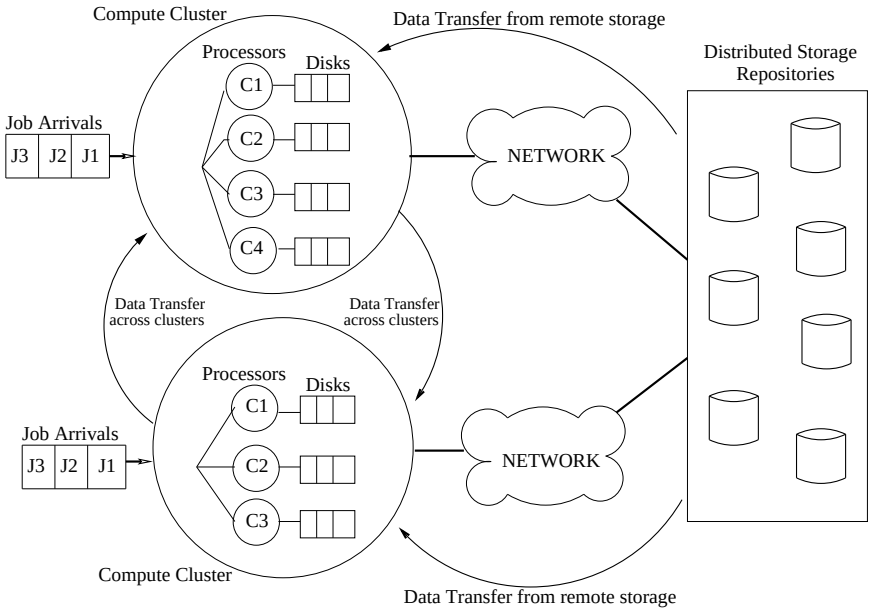


Figure 2.1: An overview of a multi-site distributed data-intensive computing scenario.

additional scientific data. Such a system model conforms to a distributed data-intensive computing paradigm where a set of computational resources, storage resources and network resources are used in a collaborative fashion to facilitate efficient execution of such data-intensive jobs.

A Motivating Example

The DZero collider detector experiment [3] at the Department of Energy's Fermi National Accelerator Laboratory is one such example of a worldwide collaboration of physicists conducting research on the fundamental nature of matter. The goal in DZero is to provide a distributed data-intensive computing system consisting of geographically distributed shareable compute and storage resources which can be used to analyze and interpret petabytes of simulation data to extract interesting physics results. The DZero experiment involves three stages. The first stage is data

reconstruction, where raw event data is converted into a format that is more amenable to being mapped to physics concepts like spin, charge etc. This is essentially a preprocessing step to make the data suitable for further analysis. The second stage involves the production of Monte Carlo simulation events to correct errors in the setup, which could be caused by the geometry of the instrumentation or particle detection efficiencies. The final stage corresponds to data analysis, which involves selecting events with certain characteristics with the goal of extracting useful scientific information. The Large Hadron Collider (LHC) [5] at the CERN laboratory is another experiment which involves analysis of several petabytes of geographically distributed data. It will be operational in 2007 and is going to employ four experiments (i.e., ATLAS, CMS, LCHb and ALICE) that involve data reconstruction and analysis jobs similar to DZero [3]. These experiments are expected to produce a petabyte of data every year. The raw data is generated at CERN, whereas the analysis jobs will typically be run at remote locations globally distributed throughout the world. This will involve staging of data from remote storage archives to the global locations where computation will take place. Addressing the scheduling issues under this model of job execution is our primary motivation. Figure 2.1 depicts an illustration of the aforesaid job scheduling scenario.

2.3 Job Scheduling

In this section, we discuss the state of the art of the job scheduling for both compute-intensive and data-intensive jobs. We highlight the inadequacies of the current job scheduling systems when applied to the data-intensive context. We then discuss some of current literature in the data-intensive scheduling domain.

2.3.1 Compute-Intensive Job Scheduling

Traditional batch job schedulers are designed for compute-intensive jobs running at supercomputer centers. They take into account CPU related metrics (e.g., user estimated job run times) and system state (e.g., queue wait times) to make scheduling decisions, but they do not take into account data related metrics. Some common examples of such batch queuing systems include the Simple Linux Utility for Resource Management (SLURM) [76], the Portable Batch System (PBS) [40], Load Sharing Facility (LSF) [77] etc.

When several data-intensive jobs are submitted to such a high-performance system, they must be executed on a subset of the processing elements. Unlike traditional compute intensive jobs, data intensive jobs may require access to a large number of files and high volumes of data. When mapping such data-intensive jobs to compute nodes, scheduling mechanisms need to take into account the overheads of retrieving files required. Moreover, the staging/transfer of files should be carefully coordinated to minimize I/O overheads.

However, compute-intensive job schedulers provide very simple mechanisms to incorporate the file staging requirements of jobs. For example, PBS has no mechanism to become aware of time required to copy remote files requested by a job to a particular compute resource. Either the batch script must explicitly stage the file to the local nodes, or the user must manually copy the files to specific nodes before submitting the job. Actual job execution can not begin until the files have been transferred. In the first case, file staging does not occur until the job is launched by the scheduler. In this scenario compute resource remain ideal while data is being transferred, wasting valuable compute resources. On the other hand, if the files are staged-in well in

advance before the job is ready to run, then these files may compete with other running jobs or ready-to-run jobs for the disk space on the compute resource. In both scenarios, the scheduler is not able to take advantage of possible data affinity created by the prior execution of jobs from the same or a different user. More intelligent data staging and scheduling algorithms could potentially reduce wasted compute cycles and make use of file affinity when executing jobs.

2.3.2 Data-Intensive Job Scheduling

Relatively little research so far has addressed the scheduling in the context of data intensive jobs. Ranganathan et al. [65] propose a decoupled approach to scheduling of computations and data for data-intensive applications and evaluate it using simulations. Their work is motivated in the context of running independent loosely coupled data-intensive jobs on data grids. The algorithm combines a scheduling scheme, called Job Data Present with a replication heuristic, referred to as Data Least Loaded in a decoupled fashion. The algorithm incorporates a notion of eligible nodes for each job, which are the set of nodes that store the file required by the job. It works by picking a job from a FIFO queue and assigning it to the node that already has the required data. If more than one compute nodes are eligible candidates, then it chooses the least loaded node. The replication mechanism Data Least Loaded is decoupled from the scheduling policy. The replication mechanism keeps track of the popularity of files, and when the popularity of a file exceeds a threshold, then the file is replicated to the least loaded node in the compute cluster. As the replication threshold decreases, the number of dynamic data replications increase. This in turn increases the possibility of increased end-point contention on the storage cluster. While the

decoupled approach of Ranganathan et al. [65] is well suited to an online environment where jobs arrive over time and there is a lack of knowledge about file access patterns of future jobs, we show that our proposed coordinated job scheduling and data replication approaches are more effective in a batch context. Phan et al. [63] also show that a coupled approach to job scheduling and data replication can yield better performance as compared to the decoupled schemes. They propose a genetic algorithm which simultaneously solves for job scheduling and data replication assignments and evaluate it using simulations. Their results show that the genetic search approach gives 20-45% improvement over greedy schedulers.

Kaya and Aykanat have developed an iterative improvement based heuristic for scheduling jobs sharing files on heterogeneous systems [45]. This work assumes a central master file server. The application is modeled as a hypergraph in a way that balances the computational load across processors. Casanova et al. [24] modified the MinMin, MaxMin, and Sufferage heuristics to take into account the additional constraint of inter-job file affinities. Their work targets the scheduling of parameter sweep applications in a Grid environment. Takefusa et al. [68] propose job scheduling and data replication policies for Data Grids and evaluate it using simulations. Their results show that a combination of OwnerComputes job scheduling strategy in conjunction with data replication policies based on number of accesses result in the minimum execution time for a job.

Jain et al. [43] model the scheduling problem as a bipartite graph coloring problem with two separate sets of vertices namely, disks and processors. In our work, in contrast, we consider grouping and mapping of jobs to compute nodes in tandem with ordering of jobs and scheduling of remote I/O operations for file transfers. Mohamed

et al. [61] presented a Close-To-Files (CF) job placement algorithm which tries to place jobs on clusters with enough idle processors that are close to the storage sites where the files reside.

Giersch et al. [38] address the problem of scheduling a collection of jobs sharing files onto heterogeneous clusters. They propose an extension to the Min-Min heuristic to lower the cost of scheduling while achieving scheduling quality (i.e., batch execution time) similar to that of Min-Min. However, their approach does not consider the global view of the jobs and their file sharing behavior while making scheduling decisions.

Desprez et al. [32] proposed an algorithm that combines data management and scheduling using a steady state approach. In their model, it is assumed that the aggregate available disk space over all compute nodes is adequate to hold at least one copy of all the files needed by the set of jobs to be scheduled. The work of Bent et al. [17] also focuses on the problem of coordination of data movement and computation scheduling. However, they assume that a job accessing multiple files can be split into a set of sub-jobs accessing a single file each, that can be allocated and scheduled independently.

Multi-query workloads also arise in the context of database applications. The work of Mehta et al. [60] is one of the first to address the problem of scheduling queries in a parallel database by considering batches of queries. Andrade et al. [12], propose a dynamic scheduling model for multi-query workloads in data analysis applications. The goal is to maximize data and computation reuse and concurrent execution on

SMP nodes through semantic caching and ordering of queries based on priority metric. These strategies mainly target efficient reuse of results from previously executed queries.

Kotz et al. [54] propose a technique called disk-directed I/O to organize multiple overlapping I/O operations with a view to optimize disk performance which is the bottleneck. The work of Kavas et al. [44] focuses on loading of executables on the compute nodes and not just data. They propose reliable multicast mechanisms to load a file to multiple nodes at once thereby reducing the storage node overheads.

Leblanc et al. [59] addresses the problem of loop scheduling on shared-memory multiprocessors. Their work is in the context of applications with data reuse, particularly the ones that employ iterative algorithms where a parallel inner loop is nested within a sequential outer loop. They propose a loop scheduling algorithm which tries to achieve load balance and cache use maximization across the processors on a shared-memory system. The key idea is to divide the inner loop parallel iteration space into chunks of iterations across processors and ensuring that for every outer loop iteration, a particular iteration of the inner loop is scheduled on the same processor to exploit cache locality. Load imbalance is handled by dynamic movement of iterations across processors. In essence, the data locality that is exploited in this work, is only due to accessing the same data again and again over multiple loop iterations spaced apart in time. In our work, we account for the fact that there is possibility of reuse among jobs which can potentially run at the same time. Parthasarathy et al. [62] proposes affinity based resource scheduling algorithms in the context of distributed data mining. They propose Distributed DOALL, which extends the previously proposed DOALL

primitive to a clustered environment. The proposed scheduling algorithms aim at maximizing the locality while trying to keep load balance.

2.4 Data Staging

The Distributed Parallel storage server(DPSS) [70], [71] is the first wide-area distributed storage system which employed optimizations like parallel TCP streams and TCP tuning. The concepts of DPSS were applied in GridFTP [11], which is a widely used protocol enabling secure, reliable and high performance data movement. It facilitates efficient data transfer between end-systems by employing techniques like multiple TCP streams per transfer, striped transfers from a set of hosts to another set of hosts and partial file transfers. It also includes optimizations like tuning of the TCP buffer size and the congestion window to improve performance. In general, GridFTP is a highly optimized data transport protocol for single file transfer. The Globus Reliable File Transfer (RFT) [9] service builds upon GridFTP and provides support for multi-file transfers. RFT focuses on issues of fault-tolerance and reliability and allows the users to monitor the progress of their file transfers. The Globus Replica location service (RLS) [29], [30] provides support for maintaining multiple replicas of each dataset. It stores a mapping of logical file names to a set of physical locations which store the file corresponding to the logical name. Globus Data Replication Service (DRS) [28] integrates the concepts of RFT and RLS in order to allow support for efficient and coordinated multi-file transfer operations.

SRB [14] is a system which provides a uniform interface to access distributed, heterogeneous storage resources. The storage resources could be anything ranging

from filesystems, databases to archival storage devices like tapes. SRB provides provides mechanisms for parallel data transfers across geographically distributed sites. It creates a number of parallel streams depending upon the availability of the network bandwidth. It also supports streaming data transfers.

Stork [53] is a specialized scheduler for data placement activities on the Grid. The scheduler allows check-pointing and monitoring of data transfers as well as use of DAG schedulers to encapsulate dependencies between computation and data movement. Stork allows the clients to specify their requirements using simple primitives which enables Stork to make decisions pertaining to how much space to allocate for a data-transfer job, what transfer protocol to use, what TCP buffer size to use etc. In our work, we focus on modeling the system topology and heterogeneity as well as the global information of a set of file requests to make efficient collective file transfer scheduling decisions. Therefore, our work is complementary to Stork and can be applied in conjunction with it.

DiskRouter [52] is a flexible infrastructure that employs main memory and disk buffering at intermediate points to speed up the data transfers. Simply speaking, it is a store and forward device that acts as a data mover between a source and destination by buffering the data sent from the sender and forwarding it to the receiver. It has dynamic flow control wherein it can slow the sender if it is running out of buffer space. It employs mechanisms like application-level overlay networks and application-level multicast to result in a more balanced network utilization. In other words, a set of disk-routers can enable the transfer of a file from a source to destination along multiple paths thereby yielding better performance. It also performs dynamic TCP buffer size tuning to maximize the utilization of bandwidth.

Internet Backplane protocol [15] allows an application to have a more finer grained control over how its data flows through the network. This is facilitated by allowing for application-managed communication buffers at various points in the network. In other words, applications have the capability to perform a coarse-grained routing of their data. IBP allows the clients to store their data in the form of IBP byte arrays at nearby nodes which can act as surrogate receivers thereby freeing the sender from the need to buffer the data. This can be used to realize lazy transmissions over the wide-area.

Swany et al. [67] exploits the "logistical effect" which essentially means improving the end-to-end performance by dividing a connection into a series of shorter, better performing connections. They address the data movement problem as finding an optimal path through the vertices of a graph. The determination of the best path through the network is done by employing a greedy, tree building method. In a recent work, Rizk et al. [66] have looked at providing TCP splitting functionality with respect to GridFTP and showed performance improvement for single file transfers. In this dissertation, we propose to optimize single file transfers by employing both multi-hop path splitting and multi-pathing in an integrated fashion. In addition, we incorporate these optimizations into a collective file-transfer scheduling framework and evaluate its effectiveness. BitTorrent is an incentive-based file sharing system which employs a tit-for-tat strategy where in the peers which contribute more data at faster rates get preferential treatment for downloads. The optimizations that we explore in this work, can be used to improve the performance of single file transfers. BitTorrent is not meant to improve the performance of single file transfers, but is more suited to a case where a bunch of peers request a set of files. Moreover, our goal is to

minimize the total transfer time in a collaborative setting where the global objective of minimizing the time is more important than each site's local benefits.

Vazkhudai et al. [74], [73] propose to use regression techniques to predict the performance of data transfers in a grid. They combine two sources of information, one being the periodic and possibly inexact network performance forecasts obtained by NWS and the other being the sporadic but exact information associated with actual GridFTP file transfers. They then propose to apply simple linear regression models on these two sources of information to predict network bandwidths. Lingyun et al. [75] proposes better estimators of future performance based on the past histories. They propose a stochastic scheduling technique which uses predicted mean and variance network information to make future predictions. These are then used to decide the amount of data to be fetched from each replica location of a requested file based on the predicted access bandwidth to retrieve the file.

A data scheduling system can accept a bunch of requests from a user-interacting system like SRB and send commands to a lower level protocol like GridFTP to perform the data transfers. The data staging algorithms proposed in this work, therefore, can be applied in conjunction with existing infrastructures like GridFTP and DiskRouter, thereby leveraging their benefits in conjunction with efficient scheduling to achieve better performance.

CHAPTER 3

SCHEDULING OF JOBS WITH BATCH-SHARED I/O

In this chapter, we address the problem of scheduling batch-shared I/O jobs in both offline and online scenarios on coupled compute and storage clusters. Batch-shared I/O simply means that the same file may be required by multiple jobs in a batch. In a coupled storage-compute cluster system, a group of machines with a large local disk pool form the storage cluster. This cluster is connected to a compute cluster over a local area network. Each compute node has one or more local disks and can request files from any of the storage nodes. Such configurations are likely to be common in institutions as well as supercomputer centers, since compute and storage clusters are designed with different goals in mind. A compute cluster will have high-end processors with high-speed networking among them. On the other hand, a storage cluster may forgo computing power in favor of large storage space. The files required by the jobs are initially resident on the storage cluster. If jobs can be executed on the storage nodes, the cost of data staging can be avoided. However, often it is not feasible to execute jobs on storage nodes – the access policies may not allow user jobs to execute on storage nodes, or the storage nodes may be designed to maximize storage space and I/O bandwidth, forgoing computation power. In this work, we assume that jobs cannot be scheduled on storage nodes – when a job is

scheduled on a processing node, the files accessed by the job must be staged on the processing node before the job is executed.

The proposed scheduling approach formulates the sharing of files (batch-shared I/O) among jobs as a hypergraph and employs a two-stage strategy for scheduling of jobs and file transfers. In the first stage, jobs are partitioned into groups via hypergraph partitioning. Each group is mapped to a compute processor in the system. In the second stage, a dynamic strategy is applied to order jobs in each group for execution and to transfer files from storage system to compute nodes for job execution. We experimentally evaluate the proposed approach using application emulators from two application domains; analysis of remotely-sensed data and biomedical imaging.

3.1 Offline Scheduling: Problem Definition

Datasets are stored, as a set of data files, on a pool of storage nodes (storage cluster). Storage nodes are connected to a pool of compute nodes (compute cluster) over a local area network. Each compute node has one or more local disks and can request files from any of the storage nodes.

A batch consists of independent sequential jobs (data analysis programs). Each job requests a subset of files in the environment and can be executed on any of the nodes in the compute cluster. Data files required by a job should be staged from the storage cluster to the compute cluster for the job to execute correctly. A data file is the unit of I/O transfer from the storage cluster to the compute cluster. The jobs in the batch may share a number of files. For example, if jobs are submitted by clients working in the same application domain, there may be a number of overlapping regions of interest, or "hot spots", as scientists in the same domain are likely to have

similar interests. Sharing of I/O also depends on how data is distributed across data files in the system. Requests by two jobs may not overlap in the underlying attribute space of the dataset, but data elements required to serve those requests might have been stored in the same set of files. If a file is required for processing by one or more jobs, it may be retrieved multiple times as a whole and transferred to the respective compute nodes.

Our objective is, given a batch of jobs and a set of files required by these jobs, to schedule the jobs in an efficient manner so as to minimize the batch execution time. Figure 3.1 depicts an illustration of this problem. Each job in the batch is represented by a compute weight, list of input files, and their file sizes.

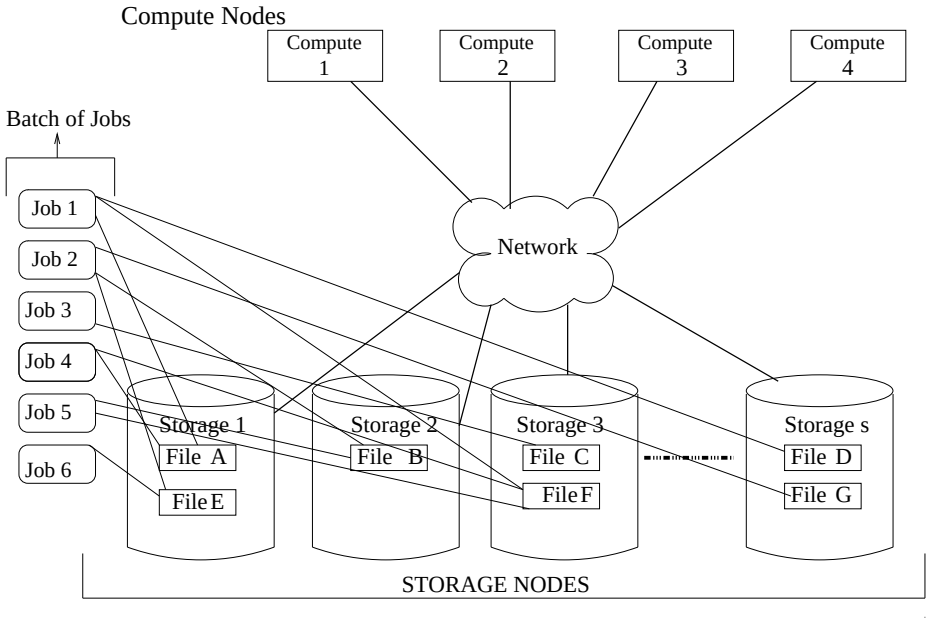


Figure 3.1: Data-Intensive job scheduling problem.

3.2 Offline Job Scheduling Strategies

In this section, we examine the MinMin, MaxMin, Sufferage, which are originally proposed for scheduling independent computational jobs to compute resources [42], along with the Shortest Job First heuristic. As in [24, 23], we modify the MinMin, MaxMin, and Sufferage to take into account 1) the time it takes to transfer input and output files to and from compute nodes in the environment and 2) files that have already been staged to a compute node in estimating the minimum completion time (MCT) of a job.

3.2.1 Shortest Job First, MinMin, MaxMin, and Sufferage

Shortest Job First (SJF)

Jobs are ordered for execution based on their expected execution times. The execution time of a job j_i is calculated as the sum of the time it takes to transfer files needed for j_i and the execution time for processing the files. In the SJF strategy, the shorter the execution time of a job is, the earlier the job is executed.

MinMin

Given a set of jobs that have not yet been scheduled, this strategy computes the MCT of each job on each idle node in the system. When computing the completion time for a job on a node, it takes into account, the files, required by the job, that is already transferred to that node by jobs previously executed on that node. Among the unscheduled jobs in the batch, MinMin chooses the job that can complete the earliest and assigns it to the node that can execute that job fastest.

MaxMin

As in MinMin, the MaxMin strategy computes the MCT of a job on each idle node in the system. Among the unscheduled jobs, it chooses the job with the maximum MCT.

Sufferage

The Sufferage strategy looks at how much a job will suffer if it is not assigned to the host that will run the job fastest. The underlying idea is that a host should execute the job that will suffer the most if the job is not assigned to that host. The sufferage of a job is computed as the difference between the job's best MCT and its second best MCT. Among the unscheduled jobs, Sufferage chooses the job with highest sufferage and assigns it to the node that will achieve the best MCT for the job.

3.3 A Hypergraph based Approach

We propose a hypergraph formulation to model sharing of files among jobs and a hypergraph partitioning based approach to compute a partitioning and mapping of jobs to compute nodes. The algorithm operates in two stages. In the first stage, we partition and map the jobs to the compute nodes. In the second stage, ordering of the jobs in each compute node is determined.

3.3.1 Hypergraph Partitioning

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. Every net $n_j \in \mathcal{N}$ is a subset of vertices, i.e., $n_j \subseteq \mathcal{V}$. The size of a net n_j is equal to the number of vertices it has, i.e., $s_j=|n_j|$. Weights

(w_i) and costs (c_j) can be assigned to the vertices ($v_i \in \mathcal{V}$) and edges ($n_j \in \mathcal{N}$) of the hypergraph, respectively. $\mathcal{P} = \{V_1, V_2, \ldots, V_P\}$ is a *P-way partition* of $\mathcal{H}$ if 1) each part is a nonempty subset of $\mathcal{V}$, 2) parts are pairwise disjoint and 3) union of P parts is equal to $\mathcal{V}$.

In a partition $\mathcal{P}$ of $\mathcal{H}$, *connectivity* λ_j of a net n_j denotes the number of parts connected by n_j. A net n_j is said to be *cut* if it connects more than one part, i.e. $\lambda_j > 1$. The cost of a partition Π is computed as

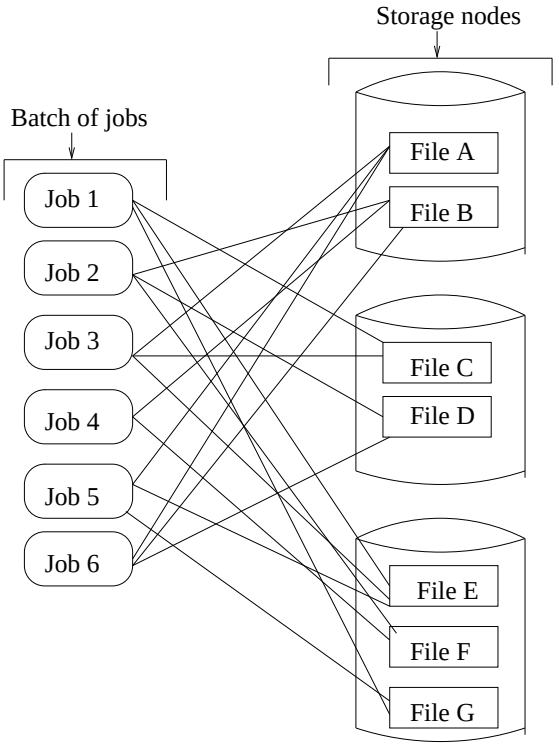
$$\chi(\Pi) = \sum_{n_j \in N_E} n_j(\lambda_j - 1) \tag{3.1}$$

where $\mathcal{N}_E$ is the set of cut nets and each cut net n_j contributes $n_j(\lambda_j - 1)$ to the cutsize. This cost metric is also known as *connectivity-1* metric. The hypergraph partitioning problem can be defined as the job of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion among the part weights is maintained. Algorithms based on the *multi-level* paradigm, such as PaToH [25], have been shown to compute good partitions quickly for this NP-hard problem.

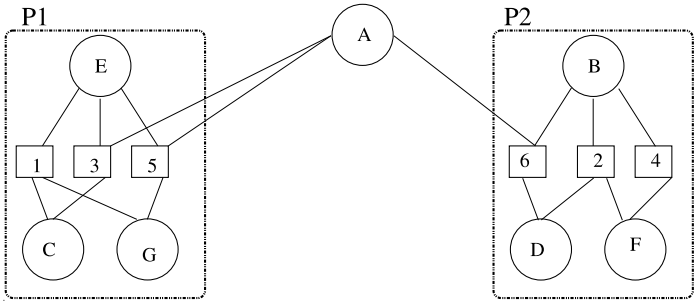
3.3.2 Hypergraph Formulation for Partitioning and Mapping of Jobs

Our goal is to partition jobs into groups such that the amount of data transfer between the storage cluster and the compute cluster is minimized while load balance across compute nodes is maintained. Our hypergraph model represents each job j_i by a vertex v_i in the hypergraph. Each hyper-edge n_j represents a file f_j and connects the vertices (jobs) that require this file as input. This hypergraph is partitioned into P groups, where P is the number of compute nodes, and each group is mapped to

a compute node. The partitioning is done so that the compute and I/O weight of the clusters are balanced and the cost of transferring shared files across clusters is minimized. Figure 3.2(b) illustrates a partitioning of the hypergraph representation of the sample batch shown in Figure 3.2(a).



a) A sample batch of jobs



b) Hypergraph representation

Figure 3.2: Hypergraph representation of a sample batch of jobs. The numbers indicate jobs. The letters are files required by the jobs.

The weight of a vertex is equal to the estimated execution time of the corresponding job. The estimated execution time of a job is calculated as the sum of I/O overhead (the transfer time of files from storage nodes plus the I/O time to read files from local disk) and the computation cost of the job. The hypergraph based strategy globally partitions all the jobs in a given batch into groups before any order for job execution is determined for a group. Hence it has to use a static vertex weights. In order to alleviate this issue and provide a better estimate of the execution time of a job, we compute the weight of a vertex as follows.

Let the set of files a job j_i needs be F_i. The cost of transferring one byte of file f_j, Tr_j, for job j_i is equal to

$$Tr_j = \frac{Prob_{FNE}}{RBW} + (1 - Prob_{FNE}) * \frac{(1 - Prob_{FE})}{RBW}. \tag{3.2}$$

Here, RBW is the I/O bandwidth between a storage node and a compute node, $Prob_{FNE}$ is the probability that job j_i will be the first job to execute in its group that requires f_j, and $Prob_{FE}$ is the probability that j_i executes on a node, to which file f_j has already been transferred. In our current implementation, we assume uniform probability distribution. Hence, we have used $Prob_{FNE} = \frac{1}{s_j}$ and $Prob_{FE} = \frac{1}{P}$. Recall that s_j is the size of the hyper-edge n_j that represents file f_j. Hence it also denotes the number of jobs that shares the file f_j. With the assumption that computation time is linear with the size of the input files, the estimated execution time of job j_i is computed as

$$ExecJ_i = \sum_{f_j \in F_i} filesize(f_j) \times (Tr_j + \frac{1}{LBW} + C) \tag{3.3}$$

where LBW is the I/O bandwidth from local disk on a compute node and C is the compute cost of one byte. By assigning the files sizes as hyper-edge costs, the proposed method reduces the job mapping problem to the P-way hypergraph partitioning problem according to the *connectivity-1* cutsize definition [25]. Each and every file needed by a job in the batch will be transferred to the compute system at least once. More specifically, if the jobs that share the file f_j is assigned to λ_j compute nodes, file f_j needs to transferred $\lambda_j - 1$ more times after its first transfer. By using expected execution times as vertex weights, the algorithm aims to balance computational load across the compute nodes.

3.3.3 Ordering of Jobs in a Group and Transfer of Files

Once the jobs are partitioned into groups, the second phase of the scheduling algorithm is to order jobs in each group and schedule transfer of files from storage cluster to compute cluster. Two jobs that are in different groups may have their input files stored on the same set of nodes. Thus, ordering of jobs in each group and transfer of files should be done in a way to minimize end-point contention on the storage cluster. We employ a strategy in which jobs within a group are scheduled based on their earliest completion time. The earliest completion time of a job is computed iteratively and dynamically based on the availability of resources.

The algorithm maintains a *Gantt chart* for storage nodes. When a job in a group is scheduled for execution, time slots are reserved on storage nodes for file transfers required for this job. These time slots for a job are marked on the Gantt chart. In calculating the duration of time slots and marking them on the Gantt chart, we assume that multiple requests to the same storage node are serialized and that a

compute node can receive a file after it has finished storing the previously received file on local disk.

The earliest completion time of a job j_i is estimated as the sum of time to stage its input files F_i and its execution time. The staging time is the time spent to make the input files ready in the compute node. If all of the input files are already in the compute node, the staging time will be zero. Otherwise, it will be the amount of time spent to transfer the last input file from the storage node. The transfer completion time for each file $f_j \in F_i$ (TCT_j) is estimated as the sum of the earliest time a transfer can start (first available slot in the Gantt chart after the time that the compute node becomes available) and the actual transfer time (size of f_j divided by the storage bandwidth; computed as the minimum of remote disk bandwidth and network bandwidth). The file f_j with the minimum TCT_j is picked and tentatively scheduled for transfer. TCTs of the rest of the input files are recomputed and the next file with the minimum TCT is picked and tentatively scheduled. This process is repeated until all of the input files are scheduled. TCT of the last file scheduled actually gives the staging time. Then the earliest estimated completion time for j_i is computed as the sum of 1) the completion time of file transfers from storage nodes, 2) I/O time to read the files on local disk, and 3) CPU time to process the files. The scheduling algorithm determines the job with the least completion time in each group, and the job j_i with the lowest *earliest completion time* out of these is scheduled first. Once j_i is scheduled, out of the other job groups (excluding the one containing j_i), the job with the minimum earliest completion time (taking into account the current reservations) is now picked and scheduled. When a running job completes, the job with earliest completion time from that group is scheduled.

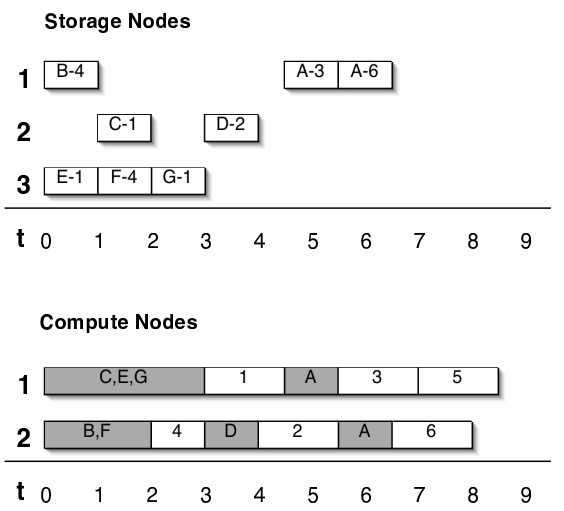


Figure 3.3: An illustration of the execution of the ordering algorithm on the batch of jobs shown in Figure 3.2.

Figure 3.3 illustrates the execution of the ordering algorithm on the batch of jobs shown in Figure 3.2. In this figure transfer of each file takes 1 unit of time, and I/O and processing of a file takes 0.3 and 0.2 units of time, respectively. Since job 4 depends on two files, its earliest completion time is 3. Hence it has been scheduled first and 1 unit of time on storage node 1 and 1 unit of time on storage node 3 have been reserved. Since a job has been scheduled from group 2, next the job with the earliest completion time from group 1 is scheduled. Since all of the jobs in the group depends on 3 files, and they can be scheduled to transfer all of the files in 3 units, we pick one of them, say job 1. The algorithm continues by reserving the transfer of files for job 1, and another job from group 2 is picked.

3.4 Experimental Results

We experimentally evaluated the scheduling algorithms using two application classes, satellite data processing and biomedical image analysis, described in Section 2.1. We used the PaToH toolkit [25] to obtain good quality partitionings of the hypergraphs generated for the workloads in the experiments. In our experiments, we observed that the hypergraph partitioning overhead is minimal compared to the execution time of a batch.

3.4.1 System Configuration

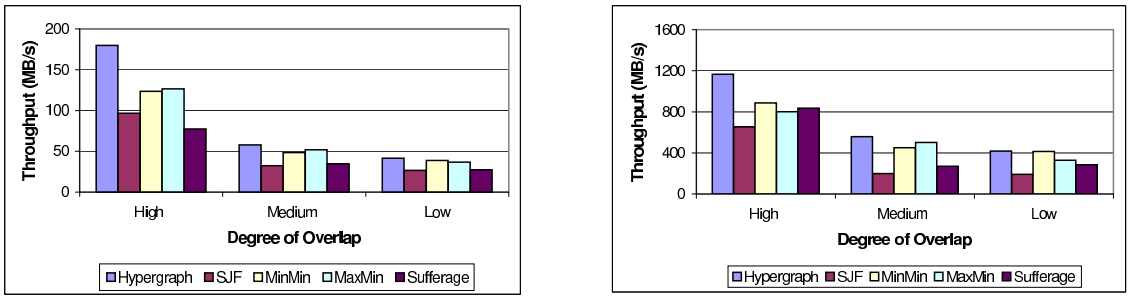
The experiments were carried out on two systems. The first system is a cluster of 933 MHz Pentium III nodes (**OSUMED**) equipped with 300GB disk space and 512MB of memory. The nodes are connected through a Switched FastEthernet. In the experiments, a subset of the nodes were designated as storage nodes, to emulate a storage cluster coupled to a compute cluster over a network. The second system (**OSC**) is a coupled compute and storage cluster system at the Ohio Supercomputer Center. The compute cluster consists of dual-processor nodes equipped with 2.4 GHz Intel P4 Xeon processors and 4 GB of memory, 62 GB of local scratch space, interconnected by an 8 Gbps Infiniband Switch. The compute cluster is connected to the storage system over another Infiniband Switch. The storage system consists of networked nodes, each of which is connected to an array of IBM FASTt600s over a Fiber Channel Switch [21]. Each node has a local file system that resides on FASTt600 storage units. For each of the workloads and hardware systems, we measured throughput (in terms of MBytes processed per second) for a batch and the amount of data transferred from storage nodes to compute nodes.

3.4.2 Application Workloads

For the satellite data processing application, we used the emulator developed in [72]. The application (**TITAN**) [27] operates on data chunks that are formed by grouping subsets of sensor readings that are close to each other in spatial and temporal dimensions. The emulator allows the user to generate datasets of varying sizes (corresponding to different numbers of days of sensor readings), the amount of data acquired per reading, and grouping of data chunks into files. In our emulation, we assigned one data chunk per file. A data analysis job specifies the data of interest via a spatio-temporal window. The corresponding files are retrieved from the storage system and processed by the data analysis job. For the image analysis application, we implemented a program to emulate studies that involve analysis on images obtained from MRI and CT scans (captured on multiple days as follow-up studies). A dataset generated by the emulator is a series of 2D images obtained for a patient and is associated with metadata describing patient and study related information (in our case, we used patient id and study id as the metadata). Each image in a dataset is associated with an imaging modality and the date of image acquisition. Each image is stored in a separate file. A data analysis program can select a subset of images based on a set of patient ids and study ids, image modality, and a date range.

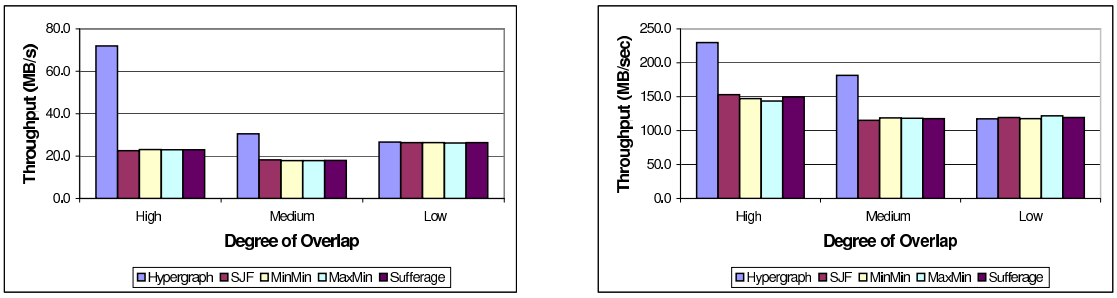
We evaluated the system for three different types of workloads; *high overlap*, *medium overlap*, and *low overlap*, each of which represents different amounts of file sharing among jobs in a batch. To generate workload for the satellite data processing application, we have simulated queries directed to geographically distant parts of the world. 4 sets of queries with 50 queries in each set have been generated representing the queries directed to 4 hot spot regions. Across the sets there is no overlap between

the queries, and in each set queries are adjusted such that for high overlap workload, they resulted in a 85% overlap, on the average, in terms of files requested by different jobs in the batch. Similarly, we generated medium and low overlap workloads with 40% and 10% overlap, respectively. For the image analysis application, different degrees of overlap is achieved by varying the values of patient and time attributes across requests by different jobs. We generated workloads with 85%, 40%, and 0% overlap for high, medium, and low overlap cases.



(a) (b)

Figure 3.4: Throughput achieved by different algorithms on the (a) OSUMED cluster and (b) OSC cluster, for the satellite data processing application.



(a) (b)

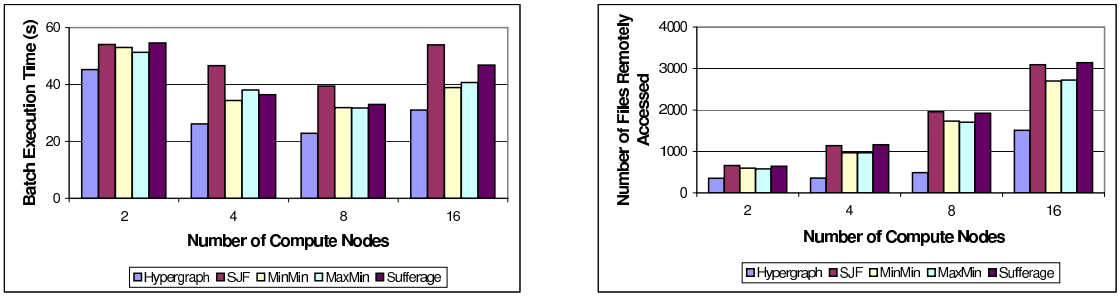
Figure 3.5: Throughput achieved by different algorithms on the (a) OSUMED cluster and (b) OSC cluster, for the biomedical image analysis application.

For the experiments, we generated 35 days worth of data, about 162 GB, for the satellite data processing application. The data was distributed across 4 storage nodes on each hardware configuration using a Hilbert-curve based declustering method [34]. Each file in the dataset was 4.5 MB. The number of jobs in a batch was equal to 200. In the high overlap case, each job accessed on an average 30 files. In the medium and low overlap cases, each job accessed on an average 8 files. For the image analysis application, the dataset generated by the emulator corresponded to a dataset of 200 patients and images acquired over several days from MRI and CT scans. The sizes of images were 1 MB and 16 MB for MRI and CT scans, respectively. The overall size of the dataset was about 68GB. Each batch comprised of 200 jobs, and each job accessed 5 MRI scans and 5 CT scans on average in the high, medium, and low overlap cases. Images for each patient were distributed among 4 storage nodes in a round robin fashion.

3.4.3 Performance Evaluation

Figures 3.4 and 3.5 show the relative performance of the various scheduling schemes on workloads with different degrees of shared I/O among jobs. These experiments were conducted using 4 compute nodes and 4 storage nodes on both OSUMED and OSC systems. As is seen from the figures, the hypergraph based strategy performs better than the other algorithms for all cases. This is because the hypergraph algorithm is able to cluster jobs that share files together, thereby reducing the number of times the same file is transferred from the remote storage system. In addition, while minimizing the networking and I/O overheads, the hypergraph algorithm maintains

computational load balance across the nodes. The gain due to hypergraph partitioning is maximum for the high overlap workload and reduces as the degree of overlap decreases, as expected. Among MinMin, MaxMin, SJF, and Sufferage, the Sufferage strategy performs slightly worse than other strategies. However, on average, MinMin, MaxMin, SJF, and Sufferage achieve more or less the same throughput irrespective of the type of workload.



(a) (b)

Figure 3.6: The performance of the scheduling strategies in the medium overlap case in the satellite data processing application as the number of compute nodes is varied on the OSC system. The number of storage nodes is equal to 4. (a) Batch execution time. (b) The number of files accessed remotely from the storage cluster.

Figure 3.6 shows how the performance of the various schemes changes as the number of compute nodes is varied on the OSC system. In this experiment, the workload for the high-overlap case in the satellite data processing application was used. The number of storage nodes was set to 4. As is seen from the figure, the hypergraph strategy achieves better performance than the other strategies in all configurations. An increase in the number of compute nodes allows for more computational parallelism. However, it also is likely to increase end-point contention on the storage nodes hence the performance decrease at 16 compute nodes in all approaches. We observe that the

volume of data transferred from the storage cluster increases with increasing number of compute nodes. This is expected since jobs will be distributed across more nodes when the number of compute nodes is increased. This will increase the probability that two jobs that share files will be mapped to different processors for execution and, as a result, the number of times a file is staged from the storage cluster to the compute cluster will increase. As is seen from the figure, the increase in the number of files transferred from storage nodes is less with the hypergraph strategy than that in the other strategies, when the number of compute nodes is increased. This is a result of the fact that sharing of files is explicitly modeled and taken into account in the hypergraph strategy for grouping and mapping of jobs to compute nodes.

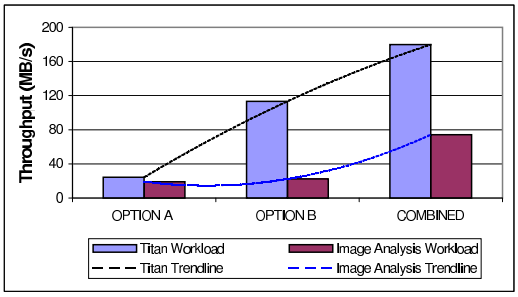


Figure 3.7: Contribution of different stages of the proposed scheduling strategy to throughput (in MBytes processed per second).

Figure 3.7 quantifies the contribution of each stage of the hypergraph partitioning algorithm. The experiments were done on the OSC system with 4 compute and 4 storage nodes for the high overlap case in both applications. *Option A* applies only the first stage (i.e., hypergraph partitioning of jobs; Section 3.3.2), but no dynamic scheduling of file transfers is done. That is, when a job is mapped to a processor, files for that job is transferred without taking into account storage node loads. *Option*

B applies only the second stage of the algorithm (Section 3.3.3) without hypergraph partitioning of jobs. The ordering of jobs is applied to the entire batch and jobs are mapped to idle processors. *Combined* is the hypergraph based scheduling strategy applying both stages. We observe that Option A does not perform as well as Option B and the combined approach. This is because, minimizing the edge-cut weight may not ensure that there is no file system contention (as different files can map to the same file system or storage node). Option B improves the performance compared to Option A in the satellite data processing application, but the performance improvement in the image analysis application is small. The best performance is obtained by the combined approach. The improvement in using the combined approach over Option B is more in the case of image analysis workload than the satellite data processing workload, since the image analysis files are larger (16 MB) in comparison to titan files (4.5 MB). In that case, the grouping and mapping of jobs to compute nodes taking into account sharing of files is more beneficial. A result of our experiments is that both grouping and mapping of jobs to compute nodes and ordering of jobs and scheduling of file transfers should be considered in tandem to obtain the best performance.

3.5 Conclusion

We presented a hypergraph based scheduling algorithm for scheduling a batch of jobs with batch shared I/O behavior on systems with coupled storage and compute clusters. The salient features of this algorithm are that 1) it formulates the sharing of files jobs as a hypergraph and uses hypergraph partitioning to map jobs to processors and 2) employs a dynamic job ordering and file transfer scheme to efficiently stage

files from storage nodes to compute nodes. Our experimental results shows that our strategy achieves better performance compared to Shortest Job First, MinMin, MaxMin, and Sufferage strategies.

3.6 Online Scheduling: Problem Definition

In this section, we address the online version of the problem definition explained in Section 3.1. For the online case, we target streams of dynamically arriving jobs which consist of independent sequential programs. Each job requests a subset of data files from a dataset and can be executed on any of the nodes in the compute cluster. The data files required by a job should be staged to the compute node where the job is allocated for it to execute correctly; a data file is the unit of I/O transfer from the storage cluster to the compute cluster. The jobs may share a number of files with previously scheduled jobs or with jobs arriving in future.

Our objective is, given a stream of dynamically arriving jobs and a set of files required by these jobs, 1) to schedule the jobs in an efficient manner, 2) to decide which files need to be remotely transferred and their respective destination nodes, so as to minimize the average job response time.

Formally, let $S =< j_1, j_2, \ldots, j_n >$ be a stream of n jobs arriving dynamically. Let $Arrival(j_i)$ be the arrival time of the job j_i and $Exec(j_i)$ be the total time the job j_i spends in execution. Some of the jobs will not be able start execution as soon as they have been submitted. Let $Start(j_i)$ be the time instant when the job j_i starts execution. In our case, this corresponds to the case when the first data transfer for the job j_i starts. If the job finds all its files locally, then it is the time when the job starts its computation. The wait time of a job $Wait(j_i)$ is the time it spends in the

queue before it starts execution.

$$Wait(j_i) = Start(j_i) - Arrival(j_i) \tag{3.4}$$

The response time $Response(j_i)$ of the job is the turnaround time which refers to the total time spent by job in the queue and in execution.

$$Response(j_i) = Wait(j_i) + Exec(j_i) \tag{3.5}$$

$Completion(j_i)$ refers to the instant when the job finishes execution.

$$Completion(j_i) = Arrival(j_i) + Response(j_i) \tag{3.6}$$

And the $AverageResponseTime$ is defined as the overall average of response times of the jobs in the stream.

$$AverageResponseTime = \frac{\sum_{i=1}^{i=n} Response(j_i)}{n} \tag{3.7}$$

3.7 Online Job Scheduling Strategies

In this section, we examine the *JobDataPresent + DataLeastLoaded* algorithm proposed in [65] in the context of data grids and the *Minimum Execution Time* (MET), *Minimum Completion Time* (MCT), *Switching Algorithm* (SA) heuristics, which were originally proposed for scheduling independent computational jobs to compute resources [58]. As in [23, 24, 51], we modify MET, MCT and SA to take into account 1) the time it takes to transfer input and output files to and from compute nodes in the environment, 2) files that have already been staged to a compute node in estimating the minimum completion time of a job and 3) in case of MCT and SA, also the files that are being staged to a compute node due to currently running job on

that node. We also integrate the Gantt chart based explicit scheduling of remote file transfers as explained in Section 3.3.3 into the MET, MCT and SA algorithms.

3.7.1 Job Data Present + Data Least Loaded, Minimum Execution Time, Minimum Completion Time, Switching Algorithm

JobDataPresent + DataLeastLoaded (JDPDLL)

The algorithm combines a scheduling scheme, called *Job Data Present* with a file replication heuristic, referred to as *Data Least Loaded* in a decoupled fashion. In this algorithm, a single file per job is employed which means that either a compute node stores the file required by a job or it does not. The algorithm incorporates a notion of eligible nodes for each job, which are the set of nodes that store the file required by the job. It works by selecting a job from a FIFO queue and assigning it to the node that already has the required data. If more than one compute node is an eligible candidate, then it chooses the least loaded node. The replication mechanism *Data Least Loaded* is decoupled from the scheduling policy. The replication mechanism keeps track of the popularity of files, and when the popularity of a file exceeds a threshold, it is replicated on the least loaded node in the compute cluster.

Minimum Execution Time (MET)

The MET heuristic assigns each job to a node that results in the least execution time ($Exec(j_i)$) for that job. As a job arrives, all the compute nodes in the cluster are examined to determine the node that gives the best execution time for the job. When computing the expected execution time of a job on a node, MET takes into account the files already available on the node. If none of the files required by a job

are found in any compute node, then the first available node is chosen to run the job. In other words, if the minimum execution time of a job an each node of the cluster is the same, then the first available node is chosen to execute the job. Therefore, MET heuristic inherently favors data locality since nodes which cache files required by a particular job are the ones which will get its best execution time.

Minimum Completion Time (MCT)

The MCT heuristic assigns each job to a node that results in that job's earliest completion time ($Completion(j_i)$). As a job arrives, all the compute nodes in the cluster are examined to determine the node that gives the earliest completion time for the job. When computing the expected completion time of a job on a node, MCT takes into account the files already available on the node and files which be available on the compute node in future due to staging of data caused by the currently executing job on the node, as well as the completion time of the currently assigned jobs to that node. Hence, MCT may discard data locality and assign a new job to node which does not have any of its files cached because the wait times on the nodes with which the job have very good file locality may be high.

Switching algorithm (SA)

The MET heuristic has a potential drawback in that it can lead to load imbalance across nodes by assigning many more jobs to some node than to others since it blindly looks at data locality without considering possible load imbalance. The MCT heuristic assigns jobs to nodes to achieve earliest completion time thereby ensuring load balance but does not necessarily exploit data locality since it may not allocate a job to a node which already has its files cached due to excess waiting times on

that node. SA heuristic is motivated by the fact that it is possible to use MET at the expense of load imbalance until a given threshold and then use MCT to smooth the load across the cluster. Similar to [58], let ib be the *load balance index* defined as $ib = load_{\min}/load_{\max}$ where $load_{\min}$ and $load_{\max}$ are the loads (completion time of the last job on that node) of minimum and maximum loaded nodes. We define two thresholds l and h. SA starts mapping jobs with MCT heuristic until the load balance index reaches to h, after that point it switches to MET and continues until load balance index decreases below l at that point it switches to MCT again and this cycle continues. In our experiments we have used $l = 0.3$ and $h = 0.7$. The goal of SA is to have a heuristic with the desirable properties of load balance as well as data locality optimization.

3.8 A Hypergraph Partitioning-based Dynamic Job Scheduling Approach

We propose an Online Hypergraph partitioning based scheduling (Online-HPS) heuristic, a two stage dynamic scheduling framework. In the first stage, jobs are mapped to compute nodes, and in the second stage, the order of the jobs in each compute node are determined. These two stages are then applied in a repeated fashion at certain scheduling events which may correspond to job arrivals or job completions. A preliminary discussion about hypergraphs and hypergraph partitioning has been described in Section 3.3.1.

3.8.1 Runtime Hypergraph-based Mapping of the System State

We develop a hypergraph formulation to model the sharing of files among the jobs present in the system. At each scheduling event, a new hypergraph is constructed which models 1) the current state of the system that includes the pending jobs and the files requested by them, 2) the currently executing jobs, and 3) the files already cached on the compute nodes due to previously executed jobs. This is followed by K-way partitioning of the hypergraph to obtain a load-balanced cut minimizing mapping of the pending jobs onto the compute nodes. The currently executing jobs are incorporated in the partitioner to take into account the current value of load on each of the compute nodes and thereby facilitate load balance as a result of the new partitioning.

Our hypergraph model consists of two sets of vertices, one set of vertices represents the pending jobs which are present in the system and the other set represents jobs currently in execution on the compute nodes. A particular job j_i is represented by a vertex v_i in the hypergraph. Each hyper-edge n_j represents a file f_j and connects to two different set of vertices, one set is the set of vertices corresponding to pending jobs that require this file as input, and the other is the vertices corresponding to running jobs which are running on a node already having a cached a copy of file f_j. This hypergraph is partitioned into P groups, where P is the number of compute nodes, and each group is mapped to a compute node. The partitioning is done so that the compute and I/O weight of the clusters are balanced and the cost of transferring shared files across clusters is minimized. The partitioning should ensure that the vertices corresponding to running jobs are allocated to the same compute

node on which they are already running. This is made sure by pinning the vertices corresponding to running jobs onto the nodes in which they are running.

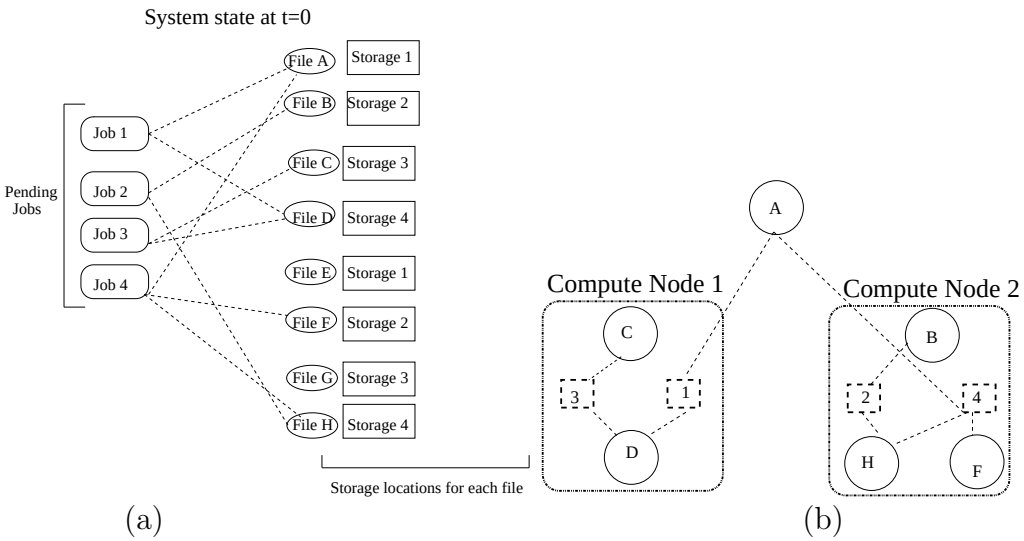


Figure 3.8: a) A snapshot of the system at t=0. Jobs 1,2,3 and 4 have arrived into the system. Letters represent files and numbers represent the jobs. Lines connecting the jobs to files represent the associated file requests for each job. b) Hypergraph partitioning across two compute nodes at t=0.

Figure 3.8(a) illustrates the state of the system at time t=0. It shows the arrival of 4 jobs into the system and their associated file requests. The boxes next to each file represent the storage locations for each file at t=0. Figure 3.8(b) illustrates a partitioning of the hypergraph representation of the system state shown in Figure 3.8(a). The figure shows that the hypergraph partitioning tries to cluster jobs sharing files together. Figure 3.9(a) illustrates the state of the system at time t=10. The figure shows two sets of vertices corresponding to pending jobs and running jobs respectively. Job 1 and Job 2 have run to completion and hence the corresponding vertices are not present. Replicas of files (i.e., multiple copies of files on the compute nodes)

have been created as files had been staged onto the compute cluster for previous jobs. The solid lines show the file requests by running jobs which can be served locally whereas the dotted lines represent the file requests which may or may not be served locally based on the result of the subsequent partitioning.

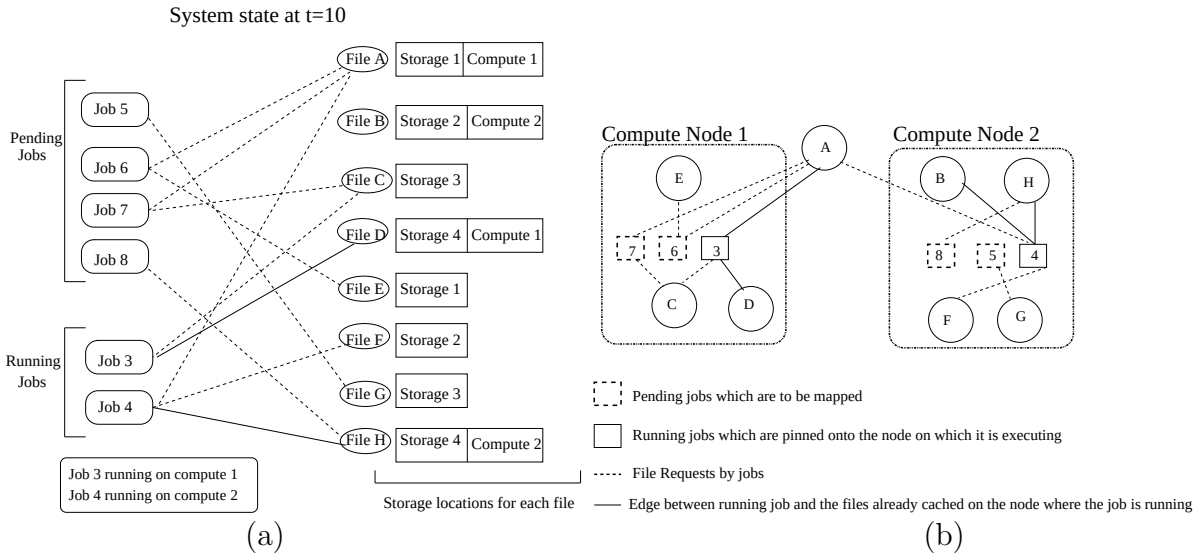


Figure 3.9: a) A snapshot of the system at t=10. Jobs 5,6,7 and 8 have arrived into the system. Jobs 1 and 2 have finished execution. Jobs 3 and 4 are currently in execution on nodes 1 and 2 respectively. b) Hypergraph partitioning across two compute nodes at t=10.

Figure 3.9(b) illustrates a partitioning of the hypergraph representation of the system state shown in Figure 3.9(a). The solid boxes represent the running jobs which have been mapped to the same nodes as in Figure 3.8(b). This is accomplished by pinning down the running jobs onto the nodes on which they are already running. The dotted boxes represent the pending jobs which have been been mapped to one of the compute nodes. The partitioning in Figure 3.9(b) shows that the jobs have been mapped to nodes with which they have strong affinity in terms of the files already

cached on those nodes while maintaining load balance. The figure shows two sets of lines. The dotted lines represent the file requests associated with the jobs. The solid lines connect each running job to the files that are already cached on the node on which the job is running. These associations between a net representing a file already cached on a node with the vertex representing the job running on that node are done to exploit the file affinities of certain pending jobs to nodes which have copies of one or more files requested by these jobs. Any pending job which requests a lot of files already cached on a node will therefore have greater inter-job affinity with the running job on that node. Therefore, in essence, we have modeled both the inter-job file sharing affinities and the job-node affinity due to caching of files.

The weight of a vertex representing a pending job is equal to the estimated execution time of the corresponding job. The estimated execution time of a job is calculated as the sum of I/O overhead (the transfer time of files from storage nodes plus the I/O time to read files from local disk) and the computation cost of the job. The hypergraph based strategy globally partitions all the existing jobs into groups before any order for job execution is determined for a group. The expected execution time of a job can possibly vary depending upon the node allocated to the job. This is because different nodes may have staged in different sets of files and therefore the job will have different locality of reference with each node. In other words, the execution times of jobs are not fixed but vary based on the allocation of the nodes and in time. The calculation of the weight of the vertices representing pending jobs is done in a similar fashion as explained in Equation 3.3. The expected execution time of a job j_i is computed as

$$EstimatedExec(j_i) = \sum_{f_j \in F_i} filesize(f_j) \times (Tr_j + \frac{1}{LBW} + C) \tag{3.8}$$

The weight of a vertex representing an already running job is equal to the remaining estimated execution time of the corresponding job. This is computed in a similar fashion as explained above except that it models the fact that some of the files required by a running job may already have been staged and therefore would not contribute to its remaining execution time.

By using expected execution times as vertex weights, the algorithm aims to balance computational load across the compute nodes. The expected execution time as calculated in equation 3.3 is based on a probabilistic model for estimating the cost of file transfer which assumes a uniform distribution. In scenarios where the data-staging costs are high and much more significant as compared to the computational costs, the impact of making such an assumption could affect load balance but the overall system performance would depend more on the connectivity metric. Therefore, the impact of the inaccuracy of this assumption would be lesser in such scenarios.

3.8.2 Job Ordering in a Compute node and Scheduling of Remote file transfers

Once the jobs have been mapped to a node, the local scheduling algorithm within each compute node decides the order in which to schedule the queued jobs and their associated file transfers. When a node becomes idle, the local scheduling algorithm running at the node decides the next job to execute on that node and also decides the schedule for its remote file transfers. Two jobs that are in different compute nodes may have their input files stored on the same set of nodes. Thus, ordering of

jobs in each compute node and transfer of files should be done in a way to minimize end-point contention on the storage cluster.

We employ the Gantt-chart based strategy proposed for the offline case(Section 3.3.3) to decide the ordering of jobs and the scheduling of file transfers.

3.9 Experimental Results

We now present an experimental evaluation of the proposed strategies along with the MET, MCT, SA and JobDataPresent-DataLeastLoaded (JDPDLL) strategies. For evaluation, we used an application class: biomedical image analysis. We compared the performance of the various scheduling schemes under a varying set of scenarios covering multiple job-file sharing patterns and different distributions of job inter-arrival times.

3.9.1 Application Workloads

We employ the image analysis application for the purpose of performance evaluation of our proposed approach. For details about the application workload generator, please refer to Section 3.4.2.

We evaluated the scheduling schemes using job traces where several aspects were varied: 1) job inter-arrival rate (to vary system load), 2) extent of file sharing among jobs, 3) temporal clustering characteristics of file-sharing behavior between jobs, and 4) burstiness of job arrivals. We generated workloads with different degrees of file sharing among jobs: *high sharing*, *medium sharing*, and *low sharing*. The different degrees of sharing is achieved by varying the values of patient and time attributes across requests by different jobs. We generated workloads with 85%, 40%, and 10%

overlap, on average, in terms of files requested by different jobs in the job trace for high, medium, and low overlap cases.

The dataset generated by the emulator corresponded to a dataset of 2000 patients and images acquired over several days from MRI and CT scans. Each job on an average accessed 6 files. The number of files accessed by a job varied from 4 to 10. The sizes of images were 4 MB and 64 MB for MRI and CT scans, respectively. The overall size of the dataset was around 2 Terabytes. Images for each patient were distributed among all the storage nodes in a round robin fashion.

The image analysis application typically involve computations equivalent of two floating point operations per word. We, therefore, emulated it with 2 FP operations per word and measured that this translates to a processing time of approximately 0.001s/MB of data in our test-bed[1].

3.9.2 Modeling the Load

In traditional compute-intensive job scheduling, the offered load on the system is calculated as:

$$OfferedLoad = \frac{\sum_{\forall i} Exec(j_i) \times n(j_i)}{P \times \max_{\forall i}(Arrival(j_i))} \tag{3.9}$$

where $n(j_i)$ represents the number of nodes allocated to a job j_i, P is the number of nodes in the system. In compute-intensive job scheduling, the $OfferedLoad$ metric is entirely dependent on the job trace under consideration and is independent of the scheduling policy being employed. However, in the data-intensive scheduling scenario

[1]It can be expected that when computation time dominates the overall execution time, the traditional job scheduling strategies would work well. The CPU power and memory bandwidth are increasing very rapidly and faster than the bandwidth of I/O devices. With such a trend, the I/O cost will become more pronounced thus entailing the need to develop scheduling algorithms which target data intensive applications.

we are focusing on, the metric defined in Equation 3.9 is no longer dependent only on the job trace but is also a function of the scheduling policy. This is because in the data-intensive scenario, the job execution times are not fixed. Instead, they vary with time due to staging of files by previously run jobs and also vary based on the node allocated to the job because of varying degrees of locality. Therefore, the job execution times depend upon the scheduling policy. To address this issue, we propose the following new characterization of load which is dependent only on the characteristics of the job trace and is independent of the scheduling policy.

Let ArrivalRate be the job arrival rate in Jobs/sec. Let ServiceRate be the expected Job service rate in Jobs/sec. The expected load is defined as follows.

$$Load = \frac{ArrivalRate}{ServiceRate} \tag{3.10}$$

Let us consider a trace of N jobs, where each job has an associated set of file transfers. Let the set of files needed by job j_i be F_i.

Let $AvgExectime$ denote the average of the execution times over all the jobs.

$$AvgExecTime = \frac{1}{N} \times \sum_{\forall i} EstimatedExec(j_i) \tag{3.11}$$

The $EstimatedExec$ time is same as calculated based on the probabilistic model explained in Section 3.8.1. To achieve an overall load of 1, The time of arrival of the last arriving job $TLarrival$ in the system is calculated as follows.

$$TLarrival = AvgExectime \times \frac{N}{P} \tag{3.12}$$

To summarize, we first determine the arrival time of the last job by using the information about the files accessed by each job so as to achieve a load value of 1. We

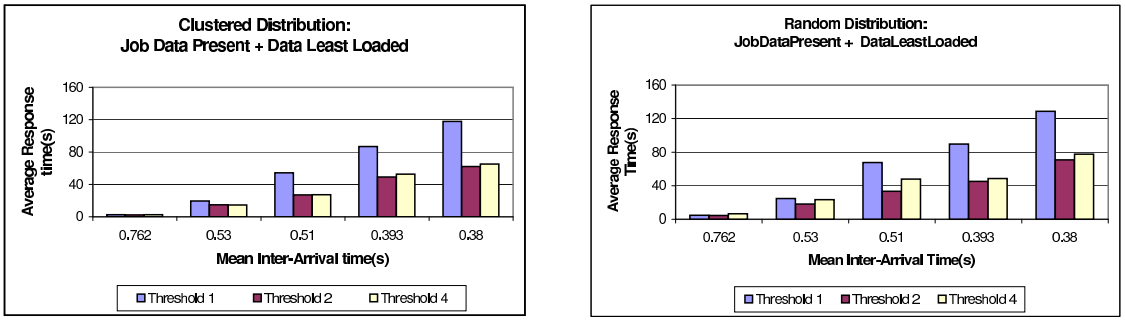
then generate job traces with different values of load by varying the number of jobs which arrive over a fixed period of time. The modeling of load is based on estimated execution times which are based on a probabilistic model as shown in equation 3.3. In reality, some jobs will require a lower actual execution time than their expected execution time if some needed files are locally available since they were staged by previously executed jobs. On the other hand, the execution time may be higher in reality, due to contention at the storage server node for file transfer.

3.9.3 Modeling the Arrival Process

We model the arrival process as a Poisson random process and evaluate it with two distributions corresponding to different job orderings - clustered distribution and random distribution. Clustered distribution refers to the case where jobs sharing files among themselves occur closer together in time. Random distribution refers to the case where jobs come in any random order. Here, the arrival times of file-sharing jobs may be widely separated from each other over time. We also model the arrival times using the model proposed by Lublin [57]. The Lublin model is based on analysis of different production logs and uses statistical methods in order to achieve a good match of synthetic traces and actual trace data. The job arrival model takes into account both the stationary arrival process during peak hours and also the daily cycle. Since the model is based on long-running jobs from production supercomputer installations, we scaled down the arrival times to reduce the overall time to run our experiments.

3.9.4 Performance Evaluation on a Cluster

We conducted our experiments using a memory/storage cluster at the Department of Biomedical Informatics at the Ohio State University. The cluster consists of 64 nodes with an aggregate 0.5 TBytes of physical memory and 48TB of disk storage. These nodes are connected to each other through Infiniband.



(a) (b)

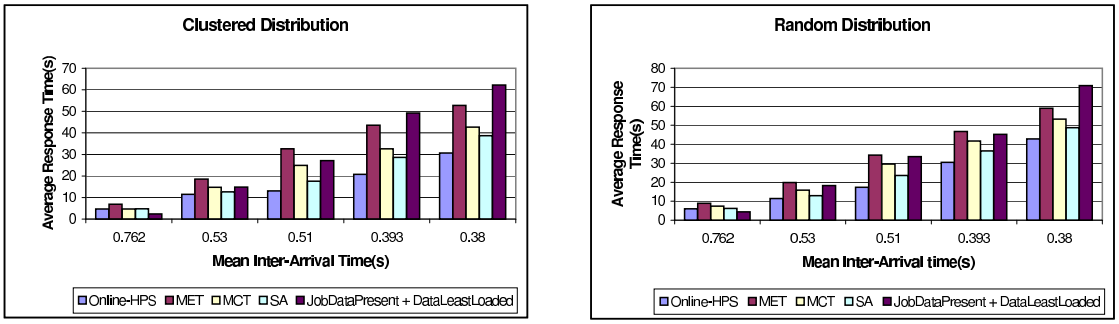
Figure 3.10: Performance of *Job Data Present* coupled with *Data Least Loaded* under various replication thresholds

One of the comparison schemes - JDPDLL - uses a critical "threshold" parameter to decide when a file should be replicated at another node. We first ran JDPDLL with different values of the replication threshold parameter. Figure 3.10 shows the variation in performance. The replication threshold represents the minimum number of references to a file by a compute node needed to trigger a replication of that file to a least-loaded node. Three different threshold values were used: 1, 2 and 4. Figure 3.10 show that the choice of the threshold has a significant effect on the performance of this algorithm - there is a trade off between benefits of increased replication and the storage node end-point contention caused by an increasing number of dynamic data replications. In our experiments, we noted that a threshold value of 2 gave the best

results and therefore this threshold is used for comparing the performance of this scheme against others.

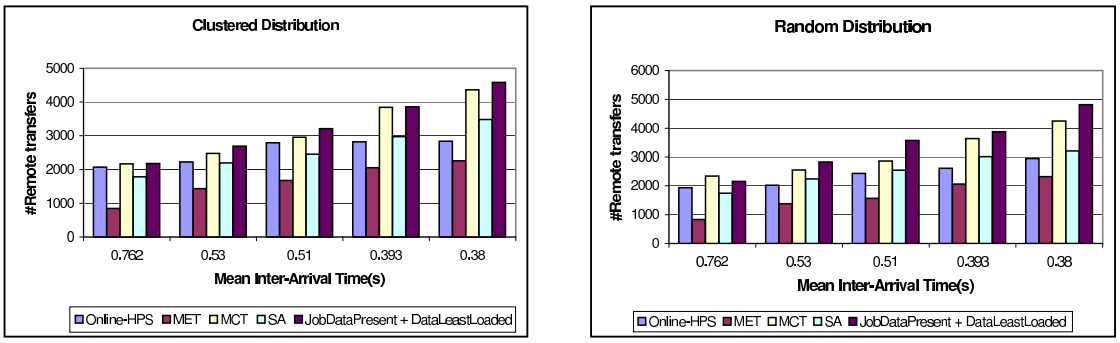
Figure 3.11 shows the relative performance of the various scheduling schemes in terms of the average response time. These experiments were conducted using 4 compute nodes and 4 storage nodes. The number of jobs in the traces used for this experiment varied from 800 to 1600 and the time of arrival of the last job in each trace was around 600 secs. The value of load based on our characterization as explained in Section 3.9.2 varied from being around 1 for the 800 job trace to around 2 for the 1600 job trace. Each compute node used for this experiment had an available space of 15GB. The figures show that hypergraph-partitioning scheme (Online-HPS) performs better than the other schemes in most of the cases. This is because it models the inter-job affinity due to file-sharing and clusters jobs that share files transfers transfer of the same file multiple times. The benefit of the proposed scheme is higher as the inter-arrival times decrease since the partitioning scheme has information about more jobs at its disposal and it exploits this information to make more informed global decisions. The base schemes MCT, MET, SA, and JDPDLL consider one job at a time when making local greedy job mapping decisions and therefore do not take into account the implicit inter-job affinities due to file sharing.

At very low loads, JDPDLL performs the best since the average inter-arrival times are high and there are significant idle periods during which file replication occurs without interfering with other file transfers. of storage node end-point of both the job play a job-inter arrival time decreases, the performance of JDPDLL deteriorates compared to Online-HPS because the file replication activity causes contention with



(a) (b)

Figure 3.11: Average Response time achieved by different algorithms for the (a) Clustered Distribution and (b) Random Distribution



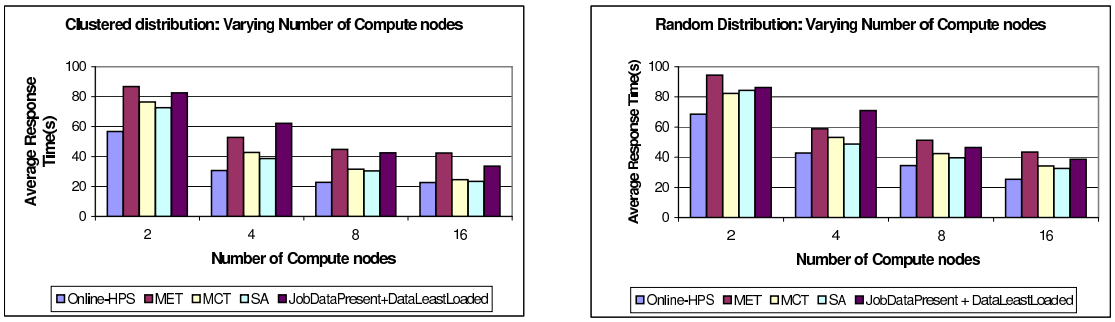
(a) (b)

Figure 3.12: Number of remote file transfers in different algorithms for the (a) Clustered Distribution and (b) Random Distribution

I/O from jobs reading input files from the storage nodes. The effect of end-point contention becomes more and more significant as the system load increases.

Figure 3.12 shows the number of remote file transfers for all the algorithms for the same set of experiments as shown in Figure 3.11. As might be expected, Online-HPS causes fewer remote transfers compared to MCT, SA and JDPDLL. This is because it attempts to cluster together jobs that share files, thereby reducing the need for

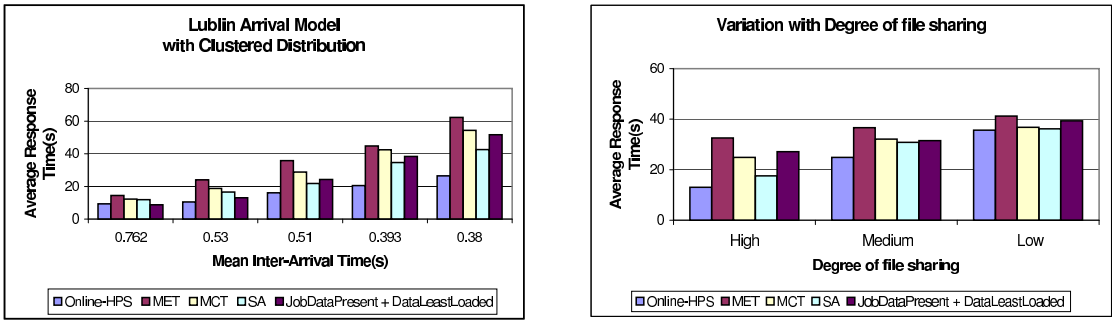
multiple transfers of the same file. The MET heuristic results in the least number of remote file transfers over all the schemes. This is because it maps each job to a node with which the job has maximum affinity in terms of the files already cached on it and required by the job. However, while doing so, it does not model the queue wait times at each node, thereby causing severe load imbalance across the nodes. Therefore, it gives the worst average response time in spite of being the best in terms of minimizing the remote file transfers.



(a) (b)

Figure 3.13: (a) Average Response time achieved by the various algorithms with varying number of compute nodes for the (a) Clustered Distribution and (b) Random Distribution

To analyze the scalability of the proposed scheme with respect to the number of compute nodes, we ran experiments with the high overlap workload consisting of 1600 jobs. The number of compute nodes were varied from 2 to 16. These experiments were run using 4 storage nodes. Figure 3.13 shows the results with varying number of compute nodes. As is seen from the figure, Online-HPS achieves the best performance in terms of average response time in all the cases.



(a) (b)

Figure 3.14: (a) Performance of the various algorithms under the Lublin arrival model and (b) Performance of the different algorithms with variation in the degree of file sharing across jobs

Figure 3.14(a) shows the relative performance of the various scheduling schemes in terms of the average response time by employing the Lublin arrival model to generate the job inter-arrival times. The results show that Online-HPS consistently performs well compared to the other schemes. The relative performance improvement under the Lublin model is higher compared to the traces modeling a Poisson arrival process. With the bursty nature of job arrival with the Lublin arrival process, the partitioning heuristic makes better job allocation decisions during bursts where a large number of queued jobs are available and inter-job file affinities can be exploited.

Figure 3.14(b) shows the relative performance of the various scheduling schemes on job traces with different degrees of shared I/O among jobs. These experiments were conducted using 4 compute nodes and 4 storage nodes. The high overlap job had 1200 jobs with an average inter-arrival time of 0.51. The medium and low overlap workloads had 800 and 400 jobs, respectively. These workloads were generated to have a uniform value of expected load. However, in reality, the medium and low overlap workloads took a longer time to execute since end-point contention became

more significant as the degree of file sharing decreased (due to increase in the number of remote file transfers). The results in Figure 3.14(b) show that the benefit of the Online-HPS scheme is greatest for the high overlap workload and reduces as the degree of overlap decreases.

3.10 Conclusion

We presented a hypergraph based dynamic scheduling heuristic for a stream of dynamically arriving independent I/O intensive jobs. The approach is based on a run-time hypergraph based modeling of the system state, followed by locality-aware and load balanced mapping and scheduling of jobs onto the compute nodes. The performance results obtained on a coupled compute/storage cluster show that it achieves significant performance improvement over previously proposed heuristics - *MET*, *MCT*, *SA* and *JobDataPresent* with *Data Least Loaded* - when there is a high degree of file sharing among jobs. The previous schemes do not explicitly consider inter-job dependencies arising out of file-sharing and thus make local decisions based on greedy heuristics. The choice of the best scheduling algorithm for a particular scenario depends upon parameters such as inter-arrival times and inter-job file sharing. Under very lightly loaded conditions, when the average job inter-arrival time is high, data replication proves to be more beneficial if a good choice of replication threshold is made. As inter-arrival times decrease, the proposed approach, which takes an integrated view of scheduling of computation and data placement, outperforms the other heuristics.

In the next chapter, we study the interplay between job scheduling and data replication, and propose algorithms which perform job mapping and data replication

in a coordinated manner. In contrast to the model discussed in this chapter, we allow explicit replication of files on compute nodes. Furthermore, we also take into account the disk space constraints on the compute node.

CHAPTER 4

COORDINATED SCHEDULING AND REPLICATION OF JOBS WITH BATCH-SHARED I/O

In this chapter, we study the interplay between job mapping and data replication. As discussed in chapter 3, whenever a job is scheduled on a processing node, the files accessed by the job must be staged on the processing node before the job is executed. In a distributed collaborative scientific computing environment, multiple scientists located at geographically different physical sites issue data staging requests which contain an overlap of requested data. In this instance, it can be possible for one of the users to retrieve the information from the other users data repository, rather then the remote storage server. The existence of multiple storage sites, therefore, entails incorporating a data replication policy which keeps multiple copies of important files in order to minimize the contention in accessing such hot-spots by multiple data requests. Executing jobs at compute sites, leads to the creation of new copies of the requested files, if not already present at the requesting location. On the other hand, replication of files to multiple compute sites drives scheduling decisions, which are trying to exploit file affinity cause by existence of the replica. Scheduling of jobs and the replication of files, therefore, share a symbiotic relationship that needs to be accounted for to ensure efficient execution of jobs. With large data files, it is

especially import to carefully consider data reuse. Due to limited storage space on the compute site, staging new files may require existing files to be evicted. The key issue is to employ an efficient caching/replacement policy which can keep popular files for a longer period of time while evicting less popular files when evictions are required.

We address this problem of coordinating scheduling and data replication for efficient execution of batch-shared I/O jobs in an offline scenario on coupled compute and storage clusters.

4.1 Problem Definition

We target batches which consist of independent sequential programs. Each job requests a subset of data files from a dataset and can be executed on any of the nodes in the compute cluster. The data files required by a job should be staged to the compute node where the job is allocated for the job to execute correctly; a data file is the unit of I/O transfer from the storage cluster to the compute cluster. The jobs in the batch may share a number of files. If a file is required for processing by one or more jobs on a particular node, it may be retrieved either from the remote storage system or from another compute node which already has the file. The decision to *replicate*[2] a file in this way depends on the mapping of the jobs that require the file and vice-versa. We assume a single port model wherein multiple requests to the same storage node are serialized and that a compute node can receive a file after it has finished storing the previously received file on local disk. We also model the case

[2]In the context of this chapter, we use the term replication to denote only the file transfers between pairs of compute nodes, one of which acts as the source and the other the destination. Staging of files from the storage system can also create replicas of files on the compute cluster. For such transfers, we use the term remote transfers and do not associate them with replication.

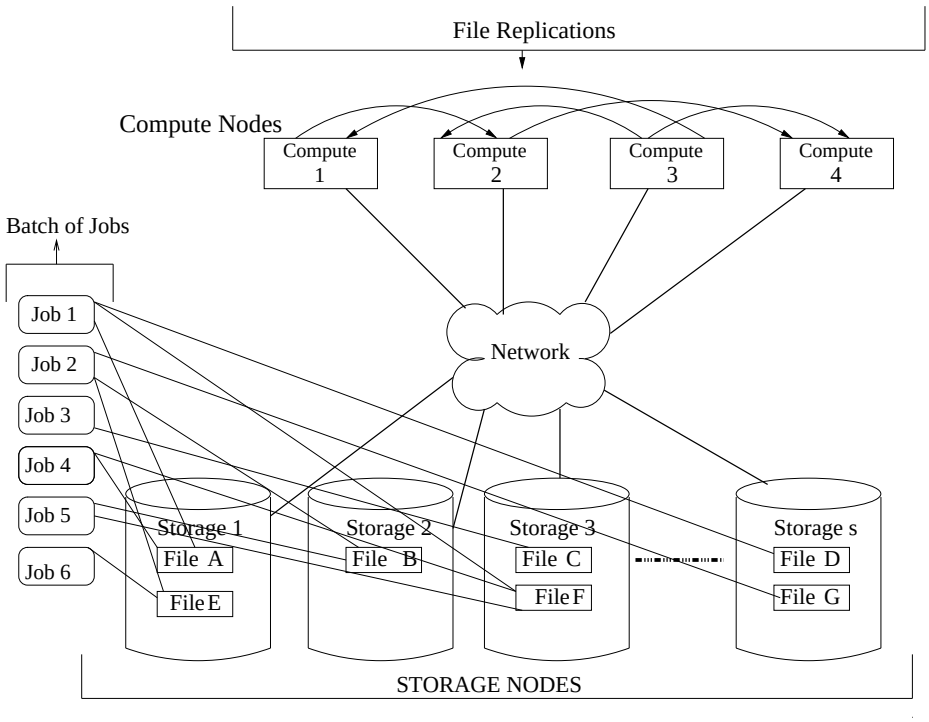


Figure 4.1: Coordinated job scheduling and data replication problem.

when the disk space on the compute cluster may not be enough to stage at least one copy of each file required by the batch at once.

Our objective is, given a batch of jobs and a set of files required by these jobs, 1) to find a mapping of jobs to nodes, 2) to decide which files need to be remotely transferred and their corresponding destination nodes, and 3) to determine which files need to be replicated and their corresponding source and destination nodes, so as to minimize the batch execution time. Figure 4.1 depicts an illustration of this problem. Each job in the batch is represented by a computation weight, a list of input files, and their file sizes.

4.2 Decoupled job scheduling and data replication

Ranganathan et. al. [65] have proposed a decoupled approach to scheduling of computations and data for data-intensive applications in a grid environment, and evaluated its effectiveness via simulation studies. The algorithm combines a scheduling scheme, called *Job Data Present* with a replication heuristic, referred to as *Data Least Loaded*, in a decoupled fashion. The details of the algorithm have been explained in Section 3.7.

4.3 Coordinated scheduling and replication: A three stage approach

We approach the problem as a three stage process. The first stage, called sub-batch selection, partitions a batch of jobs into sub-batches such that the total size of the files required for a sub-batch does not exceed the available aggregate disk space on the compute cluster. The second stage accepts a sub-batch as input and yields an allocation of the jobs in the sub-batch onto the nodes of the compute cluster to minimize the sub-batch execution time. The third stage orders the jobs allocated to each node at runtime and dynamically determines what file transfers need to be performed and how they should be scheduled to minimize the end-point contention on the storage cluster.

We propose two approaches to solve this three stage problem. The first approach formulates the sub-batch selection problem using a 0-1 Integer Programming (IP) formulation. The second stage is also modeled as a 0-1 IP formulation to determine the mapping of jobs to nodes, source and destination nodes for all replications, and the destination nodes for all remote transfers. The second approach, called BiPartition,

employs a bi-level hypergraph partitioning based scheduling heuristic that formulates the sharing of files among jobs as a hypergraph. The BiPartition strategy uses a two level partitioning approach to address the first and the second stage of the problem. The first level partitioner is used to divide the batch of jobs into sub-batches, whose data requirement fits into the available disk space on the compute cluster. The second phase yields a load balanced, cut minimizing partition of jobs on the compute cluster.

4.3.1 0-1 Integer Programming-based Approach

We approach the overall problem as a 3 stage process: The first stage is sub-batch selection; the second stage handles allocation of jobs; and the third stage implements scheduling of file transfers. We assume that each node on the compute cluster has a local disk, which can be used as a disk cache for files staged from storage nodes. We first present the IP formulation for the *unlimited disk cache space* case. In this case, each compute node has enough space to store at least one copy of each file requested by the jobs in the batch. We then describe an extension to handle limited disk cache space.

Unlimited Disk Cache Space

For this case, the sub-batch selection problem need not be solved, since the disk space on the compute cluster is not a constraint. Therefore, we directly solve for the second stage.

In the following discussion we use subscripts i and j for compute nodes, k for jobs and ℓ for files. For each job t_k the set of files accessed by that job is denoted by $Access_k$. The set of jobs accessing a particular file f_ℓ is denoted by $Require_\ell$. Let $X_{\ell i}$ be a binary variable where $X_{\ell i}$=1, if file f_ℓ is stored on node c_i, and 0 otherwise.

Let $Y_{ij\ell}$ be a binary variable where $Y_{ij\ell}$=1 if file f_ℓ on compute node c_i is replicated on compute node c_j, 0 otherwise. Let $R_{\ell i}$ be a binary variable where $R_{\ell i}$=1 if file f_ℓ is remotely transferred to node c_i, 0 otherwise. Let T_{ki} be a binary variable where T_{ki}=1 if job t_k is allocated to node c_i, 0 otherwise. The objective function is the minimization of the overall batch execution time under a set of constraints. The constraints are as follows:

A compute node can only replicate a file on another compute node if the former has the file present locally.

$$(\forall i)(\forall j, j \neq i)(\forall \ell) Y_{ij\ell} <= X_{\ell i} \tag{4.1}$$

A file is copied to (i.e., replicated on) a compute node, only if a job requiring the file is allocated to that node.

$$(\forall i)(\forall j, j \neq i)(\forall \ell) Y_{ij\ell} \leq \sum_{k \in\ Require_\ell} T_{kj} \tag{4.2}$$

The storage of a file on a node is either the result of a remote transfer or a replication.

$$(\forall i)(\forall \ell) X_{\ell i} = R_{\ell i} + \sum_{\forall j, j \neq i} Y_{ji\ell} \tag{4.3}$$

For a particular node c_i and a file f_ℓ stored on one or more other compute nodes, the file will be copied to (replicated onto) c_i from only one of the other compute nodes. Note that in the limited cache space case, a file f_ℓ may need to be copied to a node c_i multiple times, if the file is evicted due to cache space constraints and is required by jobs allocated to c_i in the future. In the unlimited cache space case, however, files are never evicted. Thus, a file is staged on a node only once. Constraint represented by equation 4.4 obeys this condition.

$$(\forall i)(\forall \ell) R_{\ell i} + \sum_{\forall j, j \neq i} Y_{ji\ell} \leq 1 \tag{4.4}$$

Each job is allocated to only a single node in the system.

$$(\forall k) \sum_{\forall i} T_{ki} = 1 \tag{4.5}$$

The allocation of a job to a node entails the staging of all the files required by the job onto the node.

$$(\forall i)(\forall k)(\forall \ell \in \ Access_k) T_{ki} \leq X_{\ell i} \tag{4.6}$$

Every file requested by the jobs in the batch will be retrieved from remote storage nodes at least once. We assume that initially all the files are resident on only the remote storage cluster. Thus, each file needs to have at least one remote transfer.

$$(\forall \ell) \sum_{\forall i} R_{\ell i} \geq 1 \tag{4.7}$$

Given these constraints, the objective is to minimize the batch execution time $Batch\_Exec\_Time$, which is the maximum of the execution time of each node. The execution time of a node c_i is defined as $Exec_i$. It is the sum of three components: the replication cost associated with that node ($Replication_i$), the computation cost of jobs allocated to that node ($Computation_i$), and the remote transfer cost of files transferred to that node ($Remote_i$).

$$Replication_i = \sum_{(\forall \ell)(\forall j, j \neq i)} (Y_{ji\ell} + Y_{ij\ell}) \times t_{rep} \times fsize(f_\ell) \tag{4.8}$$

$$Computation_i = \sum_{\forall k} Comp_k \times T_{ki} \tag{4.9}$$

$$Remote_i = \sum_{\forall \ell} R_{\ell i} \times t_{rem} \times fsize(f_\ell) \tag{4.10}$$

$$Exec_i = Replication_i + Remote_i + Computation_i \tag{4.11}$$

$$Batch\_Exec\_Time = \max_{\forall i}\{Exec_i\} \tag{4.12}$$

In these equations, t_{rep} is the replica creation cost per byte; t_{rem} is the per byte remote transfer time; $Comp_k$ represents the computation cost of job t_k, and $fsize(f_\ell)$ is the size of the file f_ℓ.

There can be overlap between communication and computation across different nodes in the system, i.e., a job may be executing on a compute node, while files for another job are being staged on another compute node. We assume a single port model wherein multiple requests to the same storage node are serialized and that a compute node can receive a file after it has finished storing the previously received file on local disk. In addition, no files are staged on a compute node while a job is executing on the node. Equation 4.11 reflects these constraints.

The IP formulation effectively exploits the global job-file sharing information and yields a one-step solution which comprises of both mapping of jobs and placement of files. However, it does not provide a solution when disk cache space is limited. To address the problem of limited cache space, we propose a 2-stage IP formulation as described in the next section.

Limited Disk Cache Space: A Two-stage 0-1 IP Solution

In the limited disk cache space case, we assume that there is enough space on each compute node to store all the files required for any single job. Since the aggregate available disk space on the compute cluster is not sufficient to stage in all the files required by the batch under consideration, a disk file eviction mechanism is needed

in conjunction with the IP based scheduling approach. The file eviction mechanism is discussed in more detail in Section 4.3.1.

The first stage of the two-stage IP solution takes as input a set of jobs and yields a subset of the jobs, referred to here as a sub-batch, such that the size of the files required for a sub-batch is less than or equal to the aggregate available disk space on the compute cluster. The second stage applies on each sub-batch the 0-1 IP formulation for the unlimited disk space with one additional constraint. The additional constraint is required to account for the fact that the space on a compute node may not be sufficient to store all the files required by the sub-batch. After a sub-batch is executed, the two stages are applied on the remaining set of pending jobs. This is repeated until all jobs in the batch are executed. We note that subsequent iterations of this two stage solution also model the fact that copies of some files have already been created on the compute cluster due to previous sub-batch executions.

First Stage: Sub-batch Selection. The primary goal of this stage is to divide a batch of jobs into subsets of jobs such that the jobs in a subset can execute on the compute cluster without the need for any file eviction. It also aims to minimize the number of sub-batches so that the scheduling overhead of multiple sub-batch executions is reduced. This is achieved by choosing a *maximally sized* subset of jobs at each sub-batch selection step, i.e., the sub-batch is formed with the maximum number of jobs which do not violate the disk space constraints. This essentially amounts to choosing a subset of jobs which have high degree of file sharing among themselves. In addition, allocation of the jobs in the sub-batch across compute nodes should be computationally balanced. The objective function of the IP formulation is

then to choose a load balanced, maximally sized subset of jobs which do not violate disk space constraints.

$$Objective\_Function = \max_{(\forall i)(\forall j)} \sum T_{ij} \tag{4.13}$$

The constraints of the IP formulation are as follows. The allocation of a job to a node entails the staging of all the files required by the job onto the node.

$$(\forall i)(\forall k)(\forall \ell \in \ Access_k) T_{ki} \leq X_{\ell i} \tag{4.14}$$

The total storage space for files stored on a particular node should not exceed the disk space available on that node.

$$(\forall i) \sum_{\forall \ell} X_{\ell i} \times fsize(f_\ell) \leq DiskSpace_i \tag{4.15}$$

A job cannot be allocated to more than one node in the system. Note that for sub-batch selection, we do not enforce the constraint that all jobs be allocated in the system, since we may be unable to do so with limited disk space.

$$(\forall k) \sum_{\forall i} T_{ki} \leq 1 \tag{4.16}$$

In order to achieve a load balanced mapping of jobs in a sub-batch to compute nodes, we enforce a constraint that the computation time on any node should be within a certain tolerance $Thresh$ of the average computation time over all the nodes. Essentially what it means is that the sub-batch should be chosen in such a way that the eventual sub-batch allocation in the second stage leads to a load balanced solution. In the following equations, Avg_Comp_time denotes the average of the computation times over all the nodes and C is the number of compute nodes.

$$(\forall i) Computation_i <= Avg\_Comp\_Time \times (1 + Thresh) \tag{4.17}$$

$$Computation_i = \sum_{\forall k} Comp_k \times T_{ki} \tag{4.18}$$

$$Avg\_Comp\_Time = \frac{1}{C} \times \sum_{\forall i} Computation_i \tag{4.19}$$

Second Stage: Sub-batch Allocation Optimized for Overall Execution Time. The sub-batch selection phase yields a subset of jobs and their allocations. However, it does not take into account the fact that some of the files may have already been copied to compute nodes (for previous sub-batches) and that fetching a file from a nearby compute node is less expensive than fetching it from the remote storage system. Thus, to achieve the best possible allocation for a given sub-batch, the set of jobs in the sub-batch is input to the 0-1 IP algorithm given in Section 4.3.1 with one additional constraint on disk space: The total storage space taken by files allocated to a particular node should not exceed the disk space available on that node.

$$(\forall i) \sum_{\forall \ell} X_{\ell i} \times fsize(f_\ell) \leq DiskSpace_i \tag{4.20}$$

This second stage of the algorithm yields a schedule and data placement information for the sub-batch.

File Eviction Policy

Once a sub-batch finishes execution, a disk file eviction mechanism is invoked, which marks files for deletion in increasing order of their popularity. At the end of this phase, each node has as much storage space as required to execute at least a single job. This phase is followed by the 2-stage process explained before - with the input being the set of remaining pending jobs. The popularity of a file, $Popularity_\ell$, is calculated as follows.

$$Popularity_\ell = \frac{Access\_Freq_\ell \times fsize(f_\ell)}{Numcopies_\ell} \tag{4.21}$$

$Access\_Freq_\ell$ represents the number of pending requests to the file. This information can be easily obtained from the original batch and the set of jobs which have already finished execution. $fsize(f_\ell)$ represents the size of the file f_ℓ. $Numcopies_\ell$ represents the number of copies of file f_ℓ in the compute cluster. If two files have the same probability of access and the same size, the file with fewer copies gets a higher popularity, since deleting that file is more likely to result in remote file transfer when the file is needed. The intuition behind including the file size in the popularity computation is that the greater the size of the file, greater the cost of getting the file back to a node. The algorithm deletes smaller files, since the cost of staging such files again in the future is lower.

4.3.2 Bi-level Hypergraph-based Approach

In recent work [55], a bi-level hypergraph partitioning based approach was developed in the context of efficient execution of parallel out-of-core applications operating on block-sparse data. We extend this idea with an intelligent method to assign weights to vertices of a hypergraph, while taking into account node-to-node data replication in the compute cluster. We also couple this bi-level job mapping approach with job ordering and file staging (see Section 3.3.3) for a complete end-to-end solution. In the bi-level hypergraph partitioning approach, the job-file sharing interactions are modeled using a hypergraph. The first level of partitioning divides the batch of jobs into multiple disjoint sub-batches such that the storage space requirement of each sub-batch does not exceed the available aggregate disk space on the compute cluster. The second level of partitioning takes as input a sub-batch and computes a load-balanced

cut minimizing mapping of the jobs in the sub-batch onto the compute nodes of the cluster.

In section 3.3.2, we discussed a hypergraph formulation which is used to obtain a load-balanced cut minimizing mapping of jobs onto nodes. To address the limited storage space constraint, we employ a different flavor of the hypergraph partitioning problem called the *Bounded Incident Net Weight (BINW) Partitioning* [55]. In BINW partitioning, the cost of a partition is again computed using the connectivity-1 cut-size definition (Eq. 3.1), but the constraint on the partitioning is different. Let $\Pi = \{V_1, V_2, ..., V_P\}$ be the P-way partition of hypergraph H and $I(V_i)$ denote the nets that are incident on vertices in V_i, i.e., $I(V_i) = \{n_j | v_k \in n_j, \forall v_k \in V_i\}$. The BINW partitioning is defined as finding a minimum cost partition where each part's incident net weight sum is bounded by a predetermined weight constraint D:

$$\sum_{n_j \in I(V_i)} c(n_j) \leq D \tag{4.22}$$

Note that P is not predetermined in this problem; however, minimizing the connectivity-1 cost while obeying the incident net weight constraint would also minimize the number of parts.

We have employed the BINW partitioner proposed in [55] as a modification of the successful multilevel hypergraph partitioner PaToH [26]. PaToH achieves P-way partitioning through recursive bisection. BINW partitioning necessitates revisiting all three phases of multilevel partitioning; *coarsening, initial partitioning* and *refinement*, as well as the recursive bisection core of PaToH. During the recursive bisection, after each bisection the nets that are in the cut are *split* into two nets in order to achieve correct accounting of the connectivity-1 cost metric. In PaToH, the default action for *size-1* nets is to discard them, since they cannot be in the cut for a future

bisection. However, since our weight constraint is based on incident net weights, we have modified the code so that the sum of the weights of such size-1 nets are accumulated in a separate weight variable for each vertex. These additional weights need to be propagated during the coarsening phase – while computing the weight constraint in the initial partitioning and refinement phases, those weights are aggregated with the sum of the internal net weights to compute a part's incident net weight.

First-level Partitioning: Sub-batch Selection

We employ the hypergraph model together with the BINW partitioning to solve the sub-batch selection problem. In our hypergraph formulation, each job t_i is represented by a vertex v_i in the hypergraph. Each hyper-edge n_j represents a file f_j and connects the vertices that require this file as input. The expected execution time of the job t_i and the size of the file f_j are used as the weights of the vertex v_i and net n_j, respectively.

Let $\Pi = \{V_1, V_2, ..., V_P\}$ be the P-way BINW partition of hypergraph H, where the weight constraint D is set to the aggregate available disk space of the compute cluster. Each partition obtained by the BINW partitioning corresponds to a sub-batch. Since the files that are required by each job is represented by nets connected to that job, the files required by the jobs of a partition V_i constitute the incident net set $I(V_i)$. By the definition of the BINW partitioning and the associated constraint (Eq. 4.22), we know that the total sum of the net weights (the sizes of the files) corresponding to files required by the partition V_i is less than D. Hence, all the files required by a sub-batch of jobs corresponding to V_i will fit into the aggregate disk space of the compute cluster. This essentially means that each sub-batch can execute on the compute cluster without disk space constraint violation. Minimizing

the connectivity-1 metric corresponds to minimizing the number of times that a file is shared among the sub-batches thereby minimizing the I/O cost because of files shared among sub-batches.

Second-level Partitioning: Job Mapping

The second level of partitioning divides a sub-batch across the nodes of the compute cluster such that the remote data transfer cost is minimized while load balance across compute nodes is maintained. We model the problem of locality-aware load-balancing as a hypergraph partitioning problem. A job is represented by a vertex and a file by a net in the hypergraph. The expected execution time of jobs and the size of files are used as weights of respective vertices and nets.

The expected execution time of a job is calculated as the sum of I/O overhead (the transfer time of files either through remote transfer or replication plus the I/O time to read files from local disk) and the computation cost of the job. To employ an existing hypergraph partitioner without any modification, we use a probabilistic approach when computing the execution time $ExecT_i$ of job t_i as vertex weights in the partitioner. Let the set of files a job t_i needs be F_i and the number of compute nodes in the system be K. The cost of transferring one byte of file f_j, Tr_j, for job t_i is equal to

$$Tr_j = \frac{Prob_{FNE}}{BW_s} + (1 - Prob_{FNE}) * \frac{(1 - Prob_{FE})}{\min(BW_s, BW_c)} \tag{4.23}$$

Here, BW_s is the minimum of I/O and network bandwidth between any storage and compute node pair, BW_c is the bandwidth between any two compute nodes of a cluster, $Prob_{FNE}$ is the probability that job t_i will be the first job to execute in its

group that requires f_j, and $Prob_{FE}$ is the probability that t_i executes on a node, to which file f_j has already been transferred. In our current implementation, we assume a uniform probability distribution, $Prob_{FNE} = \frac{1}{s_j}$ and $Prob_{FE} = \frac{s_j}{T} * \frac{1}{K}$. s_j denotes the number of jobs that share the file f_j and T denotes the number of jobs in the sub-batch. With the assumption that computation time is linearly correlated with the size of the input files, the estimated execution time of job t_i is computed as

$$ExecT_i = \sum_{f_j \in F_i} fsize(f_j) \times (Tr_j + \frac{1}{BW_\ell} + C) \tag{4.24}$$

where BW_ℓ is the I/O bandwidth from local disk on a compute node and C is the compute cost of one byte [51]. By assigning file sizes as hyper-edge weights and the estimated execution times as vertex weights, the proposed method reduces the job mapping problem to the K-way hypergraph partitioning problem according to the *connectivity-1* cutsize definition [25].

Note that the sub-batch which is given by the first level partitioner satisfies the aggregate disk space constraint on the compute cluster. A second level partitioning of such a sub-batch may lead to violation of disk space constraint on individual nodes of the cluster. To address this issue, we employ a simple heuristic. For each node of the cluster, we sort the list of files f_j to be staged onto it in the order of increasing s_j values where s_j is the number of jobs that share the file f_j. We remove files from this list in the specified order as long as the disk space requirements of the files in the list do not violate the disk space constraint. Finally, we remove any jobs assigned to this node if one or more of its files have been removed from the list. The jobs thus removed are then executed in subsequent batches.

4.4 Experimental Results

We now present an experimental evaluation of the proposed strategies along with the *Job Data Present with Data Least Loaded* approach [65] and a baseline approach based on MinMin. MinMin [58] is a well-known algorithm for job scheduling. When computing the expected minimum completion time (MCT) of a job on a node, MinMin takes into account the files already available on the node and files already available on other compute nodes which can therefore 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.

The proposed IP approach used a publicly available solver called **lp_solve** [18]. lp_solve is written in ANSI C and can be compiled on many different platforms.

4.4.1 Application Workloads

For evaluation, we used two application classes: satellite data processing and biomedical image analysis. Workload generation for the two applications has been discussed in detail in the Section 3.4.2.

We employed three different types of workloads; *high overlap*, *medium overlap*, and *low overlap*, representing different amounts of file sharing among jobs in a batch. For SAT, we simulated queries directed to geographically distant parts of the world. Four sets were generated, representing queries directed to 4 hot spot regions. Across the sets, there was no overlap between the queries, and in each set, queries were adjusted

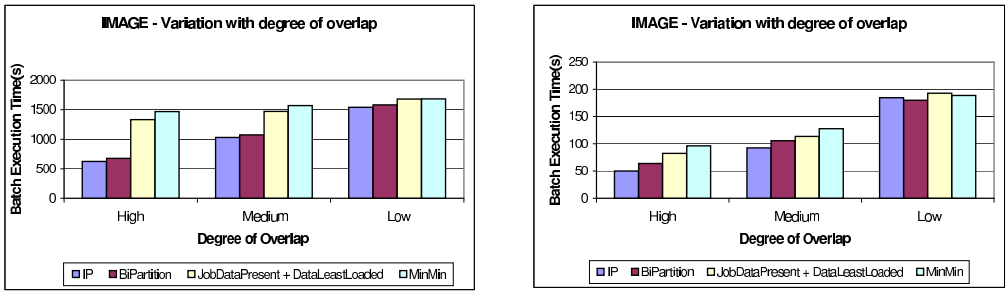
such that for high overlap workload, they resulted in 85% overlap on average, in terms of files requested by different jobs in the batch. Similarly, we generated medium and low overlap workloads with 40% and 10% overlap, respectively. For IMAGE, different degrees of overlap were achieved by varying the values of patient and time attributes across requests by different jobs. We generated workloads with 85%, 40%, and 0% overlap for high, medium, and low overlap cases, respectively.

We generated 20 days worth of data, about 50 GB for SAT. The data was distributed across the storage nodes using a Hilbert-curve based declustering method [34]. Each file in the dataset was 50 MB. In the high overlap case, each job accessed on an average 8 files. In the medium and low overlap cases, each job accessed on an average 14 files. For IMAGE, the 2 Terabyte dataset corresponded to a dataset of 2000 patients and images acquired over several days from MRI and CT scans. Each job on an average accessed 8 files. The sizes of images were 4 MB and 64 MB for MRI and CT scans, respectively. Images for each patient were distributed among all the storage nodes in a round robin fashion. The IMAGE and the SAT application typically involve computations equivalent to two floating point operations per word and this translates to a processing time of approximately 0.001s/MB of data in our test-bed.

4.4.2 System configuration

Our experiments were carried out using two systems. The first system (**OSC**) is a coupled compute and storage cluster system at the Ohio Supercomputer Center. The compute cluster consists of dual-processor nodes equipped with 2.4 GHz Intel P4 Xeon processors and 4 GB of memory, interconnected by an 8 Gbps Infiniband Switch. The

compute cluster is connected to the storage system over another Infiniband Switch. The storage system consists of networked nodes (**XIO**), each of which is connected to an array of IBM FASTt600s over a Fiber Channel Switch [21]. Each node has a local file system that resides on FASTt600 storage units. The disk bandwidth available on these storage nodes is around 210 MB/sec. The second system employs the same compute cluster as the first but uses another cluster as a storage cluster – consisting of 933 MHz Pentium III nodes (**OSUMED**) equipped with 512MB of memory. These nodes are connected through a Switched 100 Mbps Ethernet. The disk bandwidth available on these storage nodes varies from 18 MB/sec to 25 MB/sec. The bandwidth of the shared link between the OSUMED and OSC clusters is around 100 Mbps.



(a) (b)

Figure 4.2: Batch execution time achieved by different algorithms on (a) OSUMED storage cluster and (b) XIO storage cluster, for the IMAGE application

Figures 4.2 and 4.3 show the relative performance of the various scheduling/replication schemes on workloads with different degrees of shared I/O among jobs. These experiments were conducted using 4 compute nodes and 4 storage nodes for both IMAGE and SAT. Both the IMAGE workload and the SAT workload consisted of 100 jobs each. As is seen from the figures, the IP based strategy and the BiPartition approach

performs better than the other algorithms for all the cases. This is because the IP formulation is able to leverage the global job-file affinity information by incorporating it into its goal function of minimizing the batch execution time. In addition, while minimizing the I/O overheads, the IP approach also maintains computational load balance across the nodes. The BiPartition approach tries to cluster jobs that share files together, thereby attempting to avoid the transfer of the same file multiple times. In addition, the partitioning heuristic ensures load balance across nodes. The IP based approach performs slightly better than the BiPartition approach because it solves the problem of scheduling and replication in an integrated fashion and thus it is able to explore a larger search space thereby achieving a better global solution. The benefit of the proposed approaches is greatest for the high overlap workload and reduces as the degree of overlap decreases, as expected. The base schemes do not perform as well as the two proposed approaches because they are greedy heuristics and hence make local decisions without exploiting the inter-job affinities arising out of file sharing. Among the base schemes, *Job Data Present* coupled with *Data Least Loaded* does better than *MinMin* with *Implicit Replication*. This is because the *Job Data Present* strategy favors data locality and hence is able to make good use of the data replication of popular datasets performed by the *Data Least Loaded*. For the low overlap case, the IP based scheme performs slightly worse than the BiPartition scheme for some of the experiments. In the low overlap case, end-point contention increases due to a significantly higher number of remote file transfers. In the IP approach, we obtain both the job mappings and file transfer information (where from and where to replicate) statically and then realize this solution at run-time. In BiPartition, only the job mappings are obtained statically and the decision of where to fetch the file

from for each file transfer is made dynamically at run-time. The IP based scheme however does a better global modeling of the overall problem as compared to BiPartition since it captures the differential between the remote and replica access as well as the parallelism in the system. We conjecture that the effects of contention outweigh the advantage of better modeling of the IP approach when jobs do not have affinities among themselves.

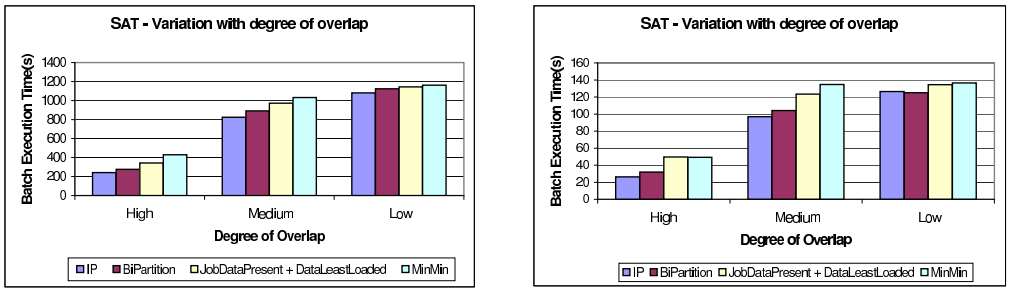
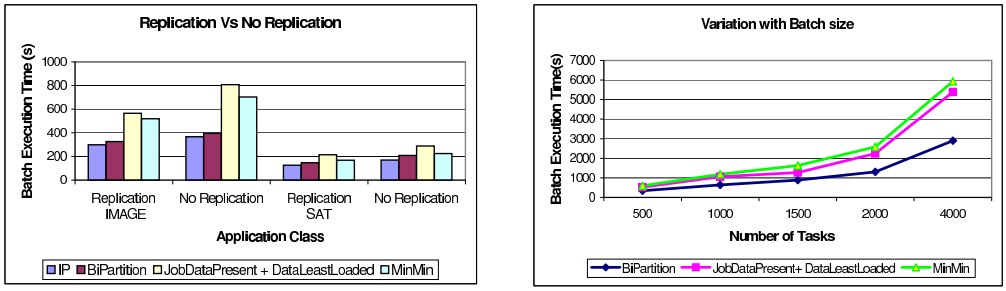


Figure 4.3: Batch execution time achieved by different algorithms on (a) OSUMED storage cluster and (b) XIO storage cluster, for the SAT application

Figure 4.4(a) quantifies the benefit which can be obtained through replication. This experiment was conducted on 8 OSC compute nodes and 4 OSUMED storage nodes. The workloads employed for both application classes was 100 job high overlap batches. The *No Replication* in Figure 4.4(a) refers to the case when there is no replication. The results show that replication gives significant performance improvement because it exploits the choice of using one of many sources of a file thereby reducing contention on the storage cluster.

Figure 4.4(b) demonstrates how the proposed scheme and the base schemes perform with respect to variation in batch size. This experiment was conducted on 4

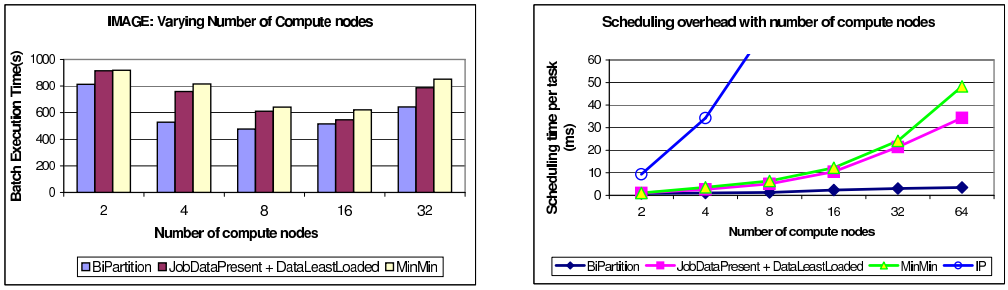


(a) (b)

Figure 4.4: (a) Benefit of compute node to compute node data replication over no replication. (b) Variation of batch execution time with increasing batch size.

OSC compute nodes and 4 XIO storage nodes using a high overlap IMAGE workload. The number of jobs in the IMAGE workload was varied from 500 to 4000. The disk space on each machine of the compute cluster is 40GB. The aggregate data requirements of the batch vary from around 40GB for the 500 job batch to around 330GB for the 4000 job batch. Here, the aggregate data requirements refer to the total disk space required to store one copy of each file. Here, we only show results for the BiPartition approach and the base schemes because for such large batch sizes, the IP based scheme has significant scheduling overhead. The results show that as the batch size increases, the base schemes show a greater increase in the batch execution time. The BiPartition scheme does the best. This is expected, since the base schemes suffer a lot of file evictions on the compute cluster as the batch size increases and the disk space becomes a constraint. The BiPartition approach makes efficient allocations of jobs and files and therefore, suffers considerably lesser evictions.

To analyze the scalability of the proposed scheme with respect to the number of compute nodes, we ran experiments with a high overlap workload consisting of 1000 high overlap IMAGE jobs. These experiments were run using 8 storage nodes of the



(a) (b)

Figure 4.5: (a) Performance of different algorithms for IMAGE with varying number of compute nodes. (b) Scheduling overhead with varying number of compute nodes.

XIO cluster. The number of compute nodes was varied from 2 to 32. Figure 4.5(a) shows the results with varying number of compute nodes. As is seen from the figure, BiPartition achieves the best performance. An increase in the number of compute nodes is likely to increase contention on the storage nodes; hence the batch execution time increases at 32 compute nodes for all approaches. We observe that the volume of data transferred increases with increasing number of compute nodes since jobs are distributed across more nodes, thereby increasing the probability that two jobs sharing files will be mapped to different nodes.

Figure 4.5(b) shows the per job scheduling times (in milliseconds) for various schemes. The graph shows that the BiPartition scheme has very little scheduling overhead. The IP based scheme has high scheduling overhead for larger configurations, due to the exponential complexity of the search. The scheduling time of *MinMin* is higher than *Job Data Present*. This is because *MinMin* involves iterating over all job-host pairs at every scheduling step. The *Job Data Present* scheme is a dynamic scheme and at each step picks up the next job from a queue and schedules it.

4.5 Conclusion

We presented two strategies for simultaneous scheduling and replication - one approach formulating the problem of coordinating scheduling and replication using a 0-1 Integer Programming and another using a bi-level hypergraph partitioning strategy. Both approaches also model disk storage space constraints at the compute cluster. The performance results show that our strategies achieve significant performance improvement over *MinMin* with *Implicit replication* and *JobDataPresent* with *Data Least Loaded*. The base schemes do not explicitly consider inter-job dependencies arising out of file-sharing and thus make local decisions based on greedy heuristics. Among the proposed algorithms, the IP formulation results in the best batch execution time. However, it suffers from high scheduling time. The BiPartition approach results in slightly longer batch execution times, but is much faster than the IP based approach. Our conclusion is that the IP based approach is attractive for small workloads, while the BiPartition approach is preferable for large scale workloads and system configurations.

In the next chapter, we propose efficient ways of scheduling file transfers in the context of data centers. We propose a scheduling heuristic that tries to maximize the parallelism and minimize the contention while scheduling the file transfers.

CHAPTER 5

COORDINATED SCHEDULING OF FILE TRANSFERS IN DATA CENTERS

In this chapter, we look at the second part of the overall mapping and scheduling problem, that is, the data staging problem. The data staging problem involves the staging of data from the simulation sites to the computational sites where the data analysis needs to be performed. In this chapter, we look at the problem in the context of data centers. Data centers consisting of collections of storage and compute clusters provide a viable environment for hosting large scientific datasets and providing analysis services. Scientific datasets are typically stored as a set of files, distributed across multiple storage nodes. Each job may request multiple files which need to be staged onto the node on which the job has been allocated. A file can be staged on a node by accessing one of the existing replicas of the file which could be present either on the storage cluster or one of the compute clusters. This should be done in a way to minimize the contention in the network or the end-points. The target environment consists of a heterogeneous collection of compute clusters connected over switched/shared network(s) to one or more storage clusters with different I/O bandwidths. We expect that such configurations will increasingly be common in supercomputing centers as the capacity of commodity disks continues to increase

and their cost per gigabyte to decrease. The topology of the network and the heterogeneity of the system govern the usefulness of each file replica for the purpose of data staging. The problem therefore amounts to scheduling of a set of file transfers to their respective destinations so as to minimize the overall completion time of file transfers.

We propose two approaches to solve the file transfer scheduling problem. The first approach formulates the problem using 0-1 Integer Programming by employing the idea of flows over time and the concept of time-expanded networks [36]. The second approach employs max-weighted graph matching to yield a schedule which tries to minimize contention and maximize the parallelism in the system. We carry out an experimental evaluation of these algorithms, comparing them against our previously proposed Insertion scheduling based heuristic [51]. Application emulators from two application domains are used - analysis of remotely sensed data and biomedical imaging.

5.1 Problem Definition

We target batches which consist of independent sequential programs. Each job requests a subset of data files from a dataset and can be executed on any of the nodes in the compute cluster. The data files required by a job should be staged to the node where the job is allocated for the job to execute correctly. The jobs in the batch may share a number of files. Therefore, a file may be required to be transferred to multiple different nodes. A file could be staged to a node by transferring it from one of its multiple possible replicas. In other words, it may be retrieved either from remote storage systems or from another compute node which already has the file. We

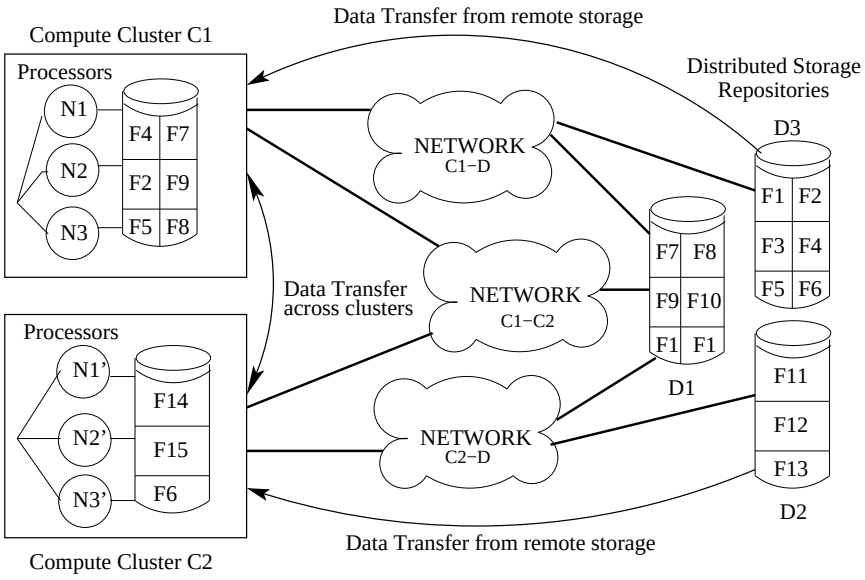


Figure 5.1: Data Staging scheduling problem.

assume a single port model wherein multiple requests to the same node are serialized and that a node can receive a file after it has finished storing the previously received file on local disk. Formally, the scheduling problem can be stated as follows:

- Given a network topology which is basically a graph $G = (V, E)$ where V represents the nodes in the system and E represents the edges connecting them. The overall topology may encompass multiple heterogeneous storage/ compute clusters but we hide the distinction between nodes belonging to different clusters by abstracting them as vertices in the overall platform graph. We assume that the Graph G is connected which means that there is a path between every pair of nodes possibly spanning multiple other nodes. Furthermore we employ a store and forward model of file transfer which implies that if a file f_ℓ needs to be transferred from a node v_i to a non-adjacent node v_j, then the file would be routed along one of the multiple possible paths between v_i and v_j. In our

model, a copy of the file is left in each intermediate node thereby increasing the number of replicas of each file leading to potentially higher parallelism for other requests.

- The input to the system is a file transfer request set represented by the set $R = \{< f_\ell, v_i >\}$, which means that the file f_ℓ needs to be transferred to the node v_i.

- The scheduling system also has access to an initial mapping of files onto the nodes. The initial mapping of files to nodes (storage and/or compute nodes, if the file has been replicated on a compute node for a previous batch) is represented by the set $D = \{< f_\ell, v_j >\}$, which means that the file f_ℓ is initially present on the node v_j.

Our objective is, given a set of file transfers where each file request is a two tuple $< f_\ell, v_i >$ *consisting of a file id and a destination node, and a network topology which is basically a graph* $G = (V, E)$ *where* V *represents the nodes in the system and* E *represents the edges connecting them, to find the complete schedule which comprises of a set of four tuples* $< v_i, v_j, f_\ell, t >$*, each tuple consisting of a source node, a destination node, a file id corresponding to the file being transferred and the file transfer start time. This has to be accomplished with the overall goal of minimizing the overall file transfer completion time.*

Fig 5.1 shows an illustration of the problem with two compute clusters and a set of distributed storage repositories. The figure shows the existence of multiple replicas of each file at different locations either of which can be used to stage the file to its associated destinations.

5.2 Problem Complexity

Definition 1. *The chromatic index of a graph $G = (V, E)$ is defined as the minimum number of colors required to color the edges of the graph such that no two adjacent edges have the same color.*

Theorem 1. *Given an arbitrary graph $G = (V, E)$, determining its chromatic number is $NP - complete$ [41].*

Theorem 2. *The optimization problem defined in Section 3.1 is $NP - complete$.*

Proof. We prove the theorem by considering a simplified version of the problem where all files are of the same size and all the links have same bandwidth. Furthermore, let us assume that each file transfer takes one unit of time to finish. We also assume that for every input request tuple $< f_\ell, v_i >$, we have determined the path of file transfer from one of the multiple possible sources of the file to the destination node. Let $Path_{\ell i}$ denote the file transfer path of the tuple $< f_\ell, v_i >$ in the set of pending input requests. $Path_{\ell i}$ consists of a set of nodes v_{i1} ,v_{i2} .... v_{ik} where the first node in the path v_{i1} is one of the multiple possible sources of file and the last node is the respective destination node for the corresponding tuple in the input request set. The optimization problem is to schedule the set of file transfers for each input request to minimize the total file transfer completion time. An edge coloring of a file transfer graph, where each edge of the graph represents a file transfer, is equivalent to generating the entire schedule since edges with the same color can start and finish at the same time. The makespan of the schedule, therefore, equals the chromatic number of the file transfer graph. In other words, the aforesaid simplified version of the problem is equivalent to finding the chromatic number of a graph. Finding the

chromatic number of an arbitrary graph $G = (V, E)$ has been proven to be $NP - complete$ [41]. Therefore, the given problem is also NP-complete. Since the simplified instance of the problem is NP-complete, the general problem is also $NP - complete$.

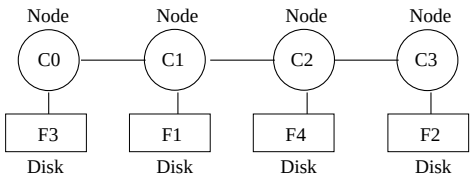
□

5.3 Data Transfer Scheduling algorithms

In Sections 5.3.2 and 5.3.3, we talk about our proposed schemes for scheduling. Section 5.3.2 proposes a 0-1 integer programming formulation of the scheduling problem. Section 5.3.3 proposes a graph matching based heuristic for the file transfer scheduling problem. In the section 5.3.1, we discuss a previously proposed scheduling scheme to solve the aforesaid scheduling problem.

5.3.1 Insertion Scheduling Based Approach

Giersch et al. [38] employ an insertion scheduling scheme to schedule file transfers. In our past work [51], we developed a Gantt chart based heuristic based on a similar idea which is applied in the conjunction with the job mapping schemes. The basic idea was to memorize the duration and the start time of file transfers for each link and use this information to generate schedules for pending requests.



Topology of the system

Fileid	Destination
1	2
1	3
2	0
4	1
3	1
3	2

File Transfer Request Set

Figure 5.2: Topology of the system and the file transfer request set.

The transfer completion time to transfer a file f_ℓ from a node v_i to a node v_j, $TCT_{\ell ij}$ is estimated as the sum of the earliest time a transfer can start and the actual transfer time (size of f_ℓ divided by the bandwidth of the link between v_i and v_j). At each step, the algorithm chooses a fileid, destination pair $< f_\ell, v_k >$ and schedules the transfer of file f_ℓ to the node v_k . To accomplish this, it finds the minimum transfer completion time TCT of each file in the input request set on its respective destination node and among them chooses the $< f_\ell, v_k >$ pair with the minimum completion time. This is accompanied by reserving time slots on the selected source of the file as well as the destination node. This process is then repeated until all the files have been scheduled to their respective destinations. Note that this is the same principle as applied in MinMin.

Fig 5.2 shows an example instance of the scheduling problem. It shows four sites connected to each other in the form of a linear topology. Each of the sites has one file initially available on it. The file transfer request set shows a set of file requests and their respective destinations. We assume that the transfer time of each file on any of the links is 1 unit. Fig 5.4 (a) shows the schedule obtained by running the Insertion scheduling algorithm on the aforesaid problem instance. The file transfer completion time obtained is 7 units of time.

Complexity Analysis of Insertion Scheduling:

Consider the topology graph $G = (V, E)$ representing the system. The input is a file transfer request set which comprises of a set of two tuples. Each two tuple $< f_\ell, v_i >$ denotes a fileid and the corresponding destination node. The computation of the minimum transfer completion time of a file on a destination node in a general topology network requires running a variant of Dijkstra's shortest path algorithm to

find which one of the multiple possible sources to stage the file from. We modify the Dijkstra's algorithm suitably to take into account the wait times of the source and the destination nodes as well as the bandwidth of the links. For a file f_ℓ under consideration, the Dijkstra's algorithm needs to be run from each of the nodes already containing the file by choosing them as source nodes for the algorithm. The complexity of Dijkstra's shortest path algorithm is $O(|E|+|V|log(|V|))$. The complexity of finding the minimum transfer completion time of a file f_ℓ is therefore, equal to $O(|V| \times (|E|+|V|log(|V|)))$. To find the $< fileid, destination >$ pair with the best transfer completion time, the computation of minimum transfer completion time has to be done for all the pending file requests. Therefore, the complexity of each step in the algorithm is $O(|R| \times |V| \times (|E \ +|V|log(|V|)))$. Finally, the complexity of the insertion scheduling algorithm is $O((|R|^2) \times |V| \times (|E|+|V|log(|V|)))$.

5.3.2 0-1 Integer Programming-based Approach

In the following discussion we use subscripts i and j for nodes, e for edges, ℓ for files and t for time. We represent time in discrete units and the smallest unit of time represents the least time taken to transfer a file from a source node to a destination node among all files and node pairs. Before we present the IP formulation, we briefly discuss the concept of the time-expanded network [36] and its construction in the context of our problem. Time-expanded networks have been defined in the context of network flows over time. The simplest version of a flows over time problem involves a network with capacities and transfer times assigned to its edges and the goal is to push the maximum amount of flow from a source node to a sink node within a given time T. A time-expanded network captures the temporal aspects of this problem in such a

manner that flows over time in the original network can be treated as just flows in the time-expanded network. Our problem has a similar flavor to a multi-commodity flow over time problem in that it involves transfer of multiple different files from multiple sources to multiple destinations so as to minimize transfer completion time.

Construction of the time-expanded network in our context.

Let F is the set of files belonging to the file transfer request set R. Let T^* denote the upper bound on the total completion time of all the file transfers. For each file $f_\ell \in F$, we construct a time expanded network $G'_l = (V'_l, E'_l)$ as follows. For each node $v_i \in V$ and each time $t = 0, ..., T^*$, we add a vertex v_{it} to the graph G'_l. For each undirected edge $e = \{v_i, v_j\}, e \in E$ connecting any two nodes v_i and v_j, $Time_{lij}$ represents the transfer time of file f_ℓ on the link $e = \{v_i, v_j\}$. For each edge $e = \{v_i, v_j\}$ and time instants t and t' such that $t' = t + Time_{lij}$, we add the directed edge $(v_{it}, v_{jt'})$ to the time expanded network G'_l if $t' \leq T^*$. Figure 5.3(a) shows the directed time-expanded network for file $F1$ for the problem instance of Figure 5.2.

0-1 IP formulation

The objective function of the IP formulation is to the minimize the overall file transfer time under a set of constraints. It solves for the following set of variables. Let $X_{\ell it}$ be a binary variable where $X_{\ell it} = 1$, if file f_ℓ is available on node v_i at time t, and 0 otherwise. Let $Y_{\ell e}$ be a binary variable where $Y_{\ell e} = 1$, if the edge e in the time expanded network G'_l is used to transfer the file f_ℓ, and 0 otherwise. Let $Busy_t$ be a binary variable where $Busy_t = 1$, if there is a file transfer which is finished at time t or a later point in time.

Constraints

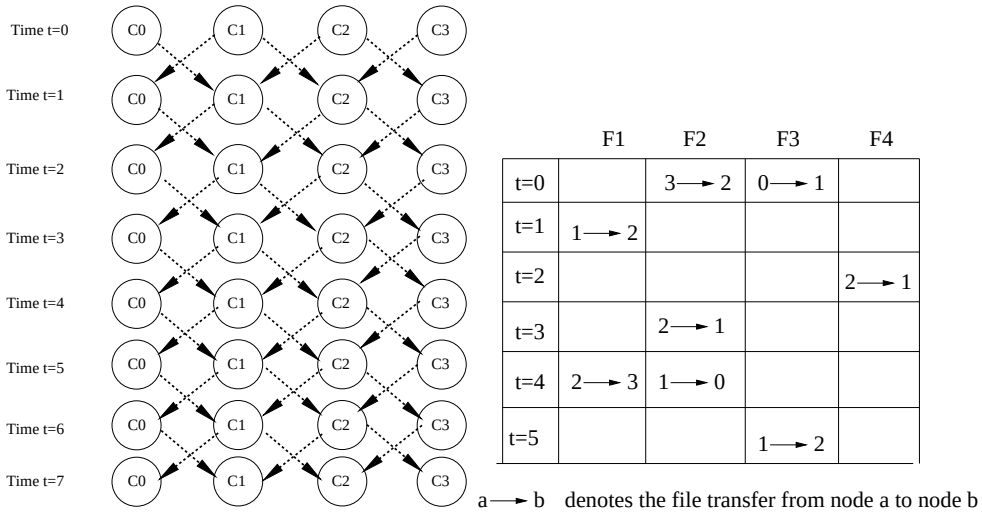


Figure 5.3: (a) Time expanded Network for file F1. and (b) Schedule obtained by the IP based approach.

At t=0, certain files are present on certain nodes.

$$(\forall \ell)(\forall i, < f_\ell, v_i > \in D) X_{\ell i 0} = 1 \tag{5.1}$$

Let $I_{\ell it}$ be the set of directed edges incident on the node v_{it} in the time-expanded network G'_l. Let $O_{\ell it}$ be the set of directed edges outgoing from the node v_{it} in the time-expanded network G'_l.

$$I_{\ell it} = (\forall j, j \neq i)(\forall t', t' \leq t)(e = (v_{jt'}, v_{it}), e \in E_{exp,l}) \tag{5.2}$$

$$O_{\ell it} = (\forall j, j \neq i)(\forall t', t' \geq t)(e = (v_{it}, v_{jt'}), e \in E_{exp,l}) \tag{5.3}$$

A file f_ℓ is present on a node v_i at time t either if its already present on the node at time $t-1$ or due to the file transfer of the file f_ℓ to the node v_i from one of the nodes v_j already having the file such that the file transfer is finished at time t.

$$(\forall \ell)(\forall i)(\forall t) X_{\ell it} = (\sum_{(\forall e, e \in I_{\ell it})} Y_{\ell e}) + X_{\ell it-1} \tag{5.4}$$

At time $t = T^*$, each file must be present at its respective destination nodes.

$$(\forall \ell)(\forall i, < f_\ell, v_i > \in R) X_{\ell i T^*} = 1 \tag{5.5}$$

A file f_ℓ can be transferred from the node v_i at time t only if its present on the node v_i at time t . In addition, atmost one outgoing arc is allowed from a node v_i at time t .

$$(\forall \ell)(\forall t)(\sum_{(\forall e, e \in O_{\ell it})} Y_{\ell e}) \leq X_{\ell it} \tag{5.6}$$

A file f_ℓ once staged to a node v_i remains available on the node.

$$(\forall \ell)(\forall i)(\forall t) X_{\ell it} \leq X_{\ell it+1} \tag{5.7}$$

The aforementioned constraints are defined for each of the time-expanded network corresponding to each unique file. The interaction between the different time-expanded networks comes from the following capacity constraints.

Each node v_i can be involved in atmost one send or receive at a time t. Let $C_{\ell it}$ be the set of all incoming and outgoing arcs of the time-expanded network G'_l that would make the node v_i busy during the time $[t, t+1)$. Note that this includes all arcs that start at time $t' \leq t$, end at a time $t' \geq (t+1)$, and having v_i as its source or target node.

$$(\forall t)(\forall i)(\sum_{(\forall \ell)(\forall e, e \in C_{\ell it})} Y_{\ell e}) \leq Busy_t \tag{5.8}$$

Objective function.

The objective is to minimize the total file transfer time $FileTransferTime$.

$$FileTransferTime = \sum_{(\forall t)} Busy_t \tag{5.9}$$

The objective function is such that the network may be busy for, say, 5 time steps with $Busy_1 = ... = Busy_5 = 1$, be idle for the next 10 time steps, $Busy_6 = ... = Busy_{15} = 0$, and finishing the transfer in the next 2 time steps, $Busy_{16} = Busy_{17} = 1$. This would lead to objective value 7, which is seemingly wrong since the network is busy even at time $t = 17$. To address this problem, we introduce the following constraint.

$$(\forall t) Busy_t \geq Busy_{t+1} \tag{5.10}$$

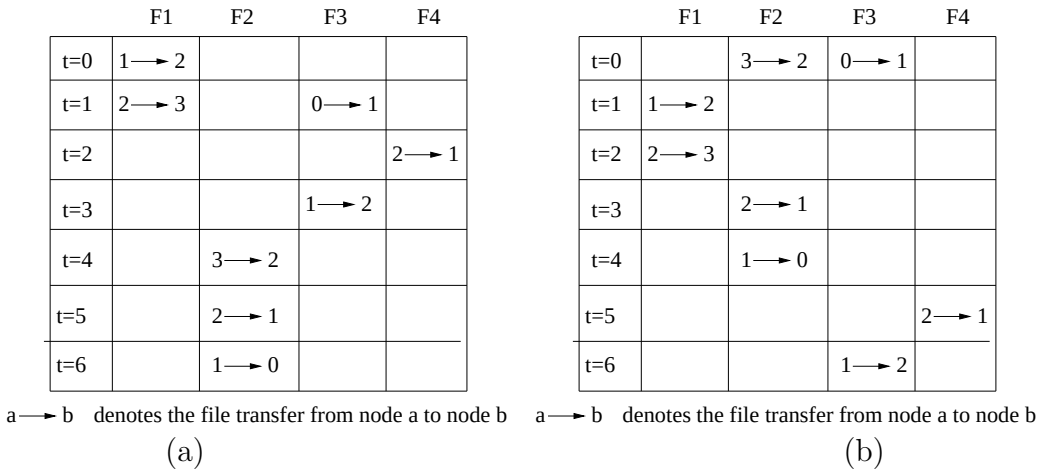


Figure 5.4: (a) Schedule obtained by Insertion scheduling. and (b) Schedule obtained by the matching based approach.

The IP formulation effectively exploits the global information comprising of the files to be transferred and the topology of the network to yield the entire schedule. Since the overall file transfer completion time obtained by the Insertion scheduling heuristic is 7 time units as shown in Figure 5.4(a), therefore we set T^* to be 7. The IP formulation gives an overall completion time of 6 units as shown by the resultant schedule in the Figure 5.3(b).

5.3.3 Max-Weighted Matching Based Scheduling Scheme (MMSS)

The MMSS is an iterative, dynamic algorithm and employs max-weighted matching heuristic as illustrated in Algorithm 5.1. For a given undirected graph $G = (V, E)$, we define the set $M \in E$ as a matching of Graph G, if no two edges in M have a common vertex. For edge-weighted graphs, the weight of the matching is the sum of the weights of the edges which form the matching. A maximum weighted matching is defined as the matching of maximum weight.

The proposed scheduling algorithm is a dynamic scheduling algorithm that uses max-weighted matching as a building block to realize the schedule. The input as mentioned in Section 5.3.1 is a file transfer request set which comprises of a set of two tuples . Each two tuple $< f_\ell, v_i >$ denotes a $< fileid, destination >$. Algorithm 5.1 outlines the matching heuristic. The goal of the algorithm is to minimize the overall file transfer completion time.

The algorithm proceeds in iterations. In each iteration, the algorithm creates a file transfer graph $G' = (V, E')$ whose vertices $v' \in V$ correspond to the nodes in the system and whose edges e' correspond to file transfers. Each input request can possibly consist of multiple hops, i.e., a set of intermediate nodes can be used to

transfer the file to its final destination. An input request $< f_\ell, v_i >$ is considered as pending, if the file f_ℓ is not yet present on the node v_i. For each such pending request, the algorithm computes the path $Path_{\ell i}$ of file transfer which yields the minimum transfer time for the file f_ℓ onto the node v_i. This step requires running a variant of Dijkstra's shortest path algorithm on the graph G to find which one of the multiple possible sources to stage the file from. We modify the Dijkstra's algorithm to take into account the wait times of the source and the destination nodes as well as the bandwidth of the links. The file transfer corresponding to the first hop of the path $Path_{\ell i}$ is then added as an edge to G' between the corresponding pair of vertices in G'. Note that for a multi-hop request, the first hop changes with time as the file gets closer to its destination node. Since multiple file transfer requests can be associated with the same source and destination node, therefore, each pair of vertices in the file transfer graph can possibly have multiple edges between them. The weight of an edge in the file transfer graph corresponding to an input request is $\frac{1}{TCT}$ where TCT is the expected minimum completion time of the request. The idea behind this weight assignment is to give higher priority to file transfers which can finish early. Finally, the algorithm employs max-weighted graph matching on the file transfer graph to obtain a set of non-contending ready file transfers and schedules them. This procedure works iteratively until all the file transfers have been scheduled.

Fig 5.4(b) shows the schedule obtained by running the matching based algorithm on the aforesaid problem instance. The file transfer completion time obtained is 7 units of time.

Algorithm 5.1 Maximum Weighted Matching based Scheduling Heuristic

Require: Topology denoted by $G = (V, E)$ and an input request set consisting of $< f_\ell, v_i >$ pairs

1: **while** there exists a pending request **do**
2: **for** each pending request $< f_\ell, v_i >$ **do**
3: Run the Modified Dijkstra's algorithm. Let $Path_{\ell i}$ denote the file transfer path which yields the earliest completion time for the request $< f_\ell, v_i >$.
4: Create a file transfer graph $G' = (V', E')$ as follows.
5: **for** each pending request $< f_\ell, v_i >$ **do**
6: Let nodes v_{i1} and v_{i2} comprise the first hop of the file transfer path $Path_{\ell i}$.
7: $V' = V' \cup \{v_{i1}, v_{i2}\}$.
8: Add an edge with weight $\frac{1}{TCT}$ between v_{i1} and v_{i2} in G'. Here, TCT denotes the minimum completion time of the request
9: Run the Max-weighted matching algorithm on the Graph G' to get a Matching
10: Schedule the chosen set of edges belonging to the Matching

Complexity Analysis of the Matching Heuristic:

Edmonds et al. [33] proposes a $O(|V|^4)$ algorithm for finding maximal matchings in graphs. We employ Gabow's implementation of the Edmond's algorithm for computing maximal matching on graphs [37]. The complexity of the Gabow's implementation is $O(|V|^3)$.

Before we go further, we analyze an iterative graph matching procedure for a Graph G. The analysis is used to compute the run-time complexity of the matching based scheduling approach.

Algorithm 5.2 shows an iterative matching algorithm which at each step, chooses a different set of edges of a Graph G such that those edges constitute a matching and marks all those edges. The set of edges $e \in E$ chosen at each step constitute disjoint sets.

Algorithm 5.2 Iterative Matching

Require: Graph $G = (V, E)$

1: Unmark all the edges of the graph G
2: **while** There exists an edge $e \in E$ which is unmarked **do**
3: Create a Graph $G' = (V', E')$ where $V' = V$ and E' consists of edges $e \in E$ such that e is unmarked
4: Run a maximal matching algorithm on the Graph G' to get a Matching M
5: Set the chosen set of edges which constitute the Matching M as marked

Theorem 3. *Given a Graph $G = (V, E)$ with atmost one edge for every vertex pair , the number of times a maximal matching algorithm needs to be run on the Graph G in order to cover all its edges is $O(|V|)$. In other words, the number of iterations of the while loop in Algorithm 5.2 is $O(|V|)$.*

Proof. Consider a specific instance where the Graph $G = (V, E)$ is a clique. A clique consists of $\frac{|V|\times(|V|-1)}{2}$ edges. The set of edges of a clique is basically the union of $|V| - 1$ sets of $\frac{|V|}{2}$ edges each such that for each edge set of size $\frac{|V|}{2}$, no two edges belonging to it share a common vertex. Each such edge set, therefore, constitutes a matching. Each step of the algorithm 5.2, therefore, chooses $\frac{|V|}{2}$ edges which form a matching. Since the number of such edge sets is $|V| - 1$, therefore, the number of steps required to cover all its edges is $|V| - 1$. For general graphs $G = (V, E)$, where $E \leq |V|^2$, the complexity can therefore be atmost $O(|V|)$.

□

Corollary 1. *Given a Graph $G = (V, E)$ with atmost $|K|$ edges for every vertex pair , the number of times a maximal matching algorithm needs to be run on the Graph G in order to cover all its edges is $O(|K| \times |V|)$.*

For a file transfer input request set R and a platform graph $G = (V, E)$, each input request can atmost require $O(|V|)$ hops in the platform graph. Therefore, the total number of file transfer edges can atmost be $O(|R| \times |V|)$. In the worst case, each file transfer request can involve the the transfer of files through the same set of edges in the graph G. Therefore, there can be atmost $O(|R|)$ file transfer request edges between any pair of vertices in the worst case. By applying the aforesaid theorem and the corollary, we can prove that the matching algorithm is run atmost $O(|R| \times |V|)$ to cover all the file transfer request edges thereby obtain the complete schedule. Therefore, the worst case complexity of the matching based scheduling heuristic is $O((|R|) \times (|V|^4))$.

The number of file transfer requests $|R|$ is typically orders of magnitude higher than the number of vertices $|V|$ in the platform graph G. Therefore, in practice, the matching based heuristic is expected to perform much faster as compared to the Insertion scheduling approach explained in Section 5.3.1.

5.4 Experimental Results

In this section, IP refers to the integer programming approach proposed in Section 5.3.2, *Matching* refers to the graph matching based approach proposed in Section 5.3.3 and *Insertion* refers to the insertion scheduling approach explained in Section 5.3.1. We implement another heuristic which acts as a baseline scheduling scheme for comparison. The heuristic is called as *Indep_Local* and is a relatively simpler scheduling scheme where each destination node knows the the set of files it needs and makes requests for each of them one by one. The destination nodes acting

as clients do not interact with each other before making their respective requests and the centralized scheduler ensures that contending requests are serialized.

The proposed IP approach used an optimization technique [35] in conjunction with the well known solver called ILOG-CPLEX [7] available through the NEOS Optimization Server [31]. For the purpose of the experiments, the upper bound on the overall file transfer completion time T^* was set to be value obtained by the *Matching* approach. For fair comparison, the scheduling time of the Integer Programming approach equals the sum of the time taken by the *Matching* approach and the time taken by the IP solver. Since the feaspump solver gives feasible solutions which need not be optimal, therefore we apply binary search in conjunction with the solver to get the optimal value of the objective function.

For evaluation, we compared the performance of the various scheduling schemes under a varying set of scenarios covering multiple job-file sharing patterns and different topologies. We consider three different kinds of topologies, namely fully connected topologies which can be composed with Infiniband fabrics, bipartite topologies and random topologies. For the workloads, we employ both randomly generated workloads as well as workloads derived from two application classes: satellite data processing and biomedical image analysis. To generate datasets for the satellite data processing application (referred to here as **SAT**), we employed an emulator developed in [72]. The application [27] operates on data chunks that are formed by grouping subsets of sensor readings that are close to each other in spatial and temporal dimensions. These chunks can be organized into multiple files. In our emulation, we assigned one data chunk per file. A satellite data analysis job specifies the data of interest via a spatio-temporal window. For the image analysis application (referred to here as

IA), we implemented a program to emulate studies that involve analysis on images obtained from MRI and CT scans (captured on multiple days as follow-up studies). An image dataset consists of a series of 2D images obtained for a patient and is associated with meta-data describing patient and study related information (in our case, we used patient id and study id as the meta-data). Each image in a dataset is associated with an imaging modality and the date of image acquisition, and is stored in a separate file. An image analysis program can select a subset of images based on a set of patient ids and study ids, image modality, and a date range.

For SAT, the 250GB dataset was distributed across the storage nodes using a Hilbert-curve based declustering method [34]. Each file in the dataset was 50 MB. For IA, the 1 Terabyte dataset corresponded to a dataset of 2000 patients and images acquired over several days from MRI and CT scans. The sizes of images were 10 MB and 100 MB for MRI and CT scans, respectively. Images for each patient were distributed among all the storage nodes in a round robin fashion.

To generate the input file request set for the two application domains, we apply the hypergraph partitioning based job-mapping technique [51] to map a batch of jobs onto a set of compute nodes. Since each job is associated with a set of files it needs, therefore, the job mapping provides information about the respective destination nodes for each file. Moreover, since the job partitioning is locality conscious, therefore, the number of different destination nodes for each file is very low.

We conducted our experiments using a memory/storage cluster at the Department of Biomedical Informatics at the Ohio State University. The cluster consists of 64 nodes with an aggregate 0.5 TBytes of physical memory and 48TB of disk storage.

These nodes are connected to each other through Infiniband. We also used simulations to understand the performance of the various scheduling schemes on different topologies. We ran our simulations using the **Simgrid Toolkit** [23, 56]. This toolkit implements event-driven simulation of applications on heterogeneous distributed systems. It models a resource by two performance characteristics: latency (time to access the resource) and service rate (number of work units performed per time unit.

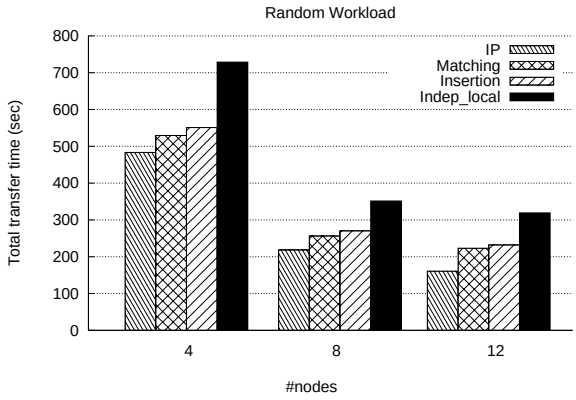
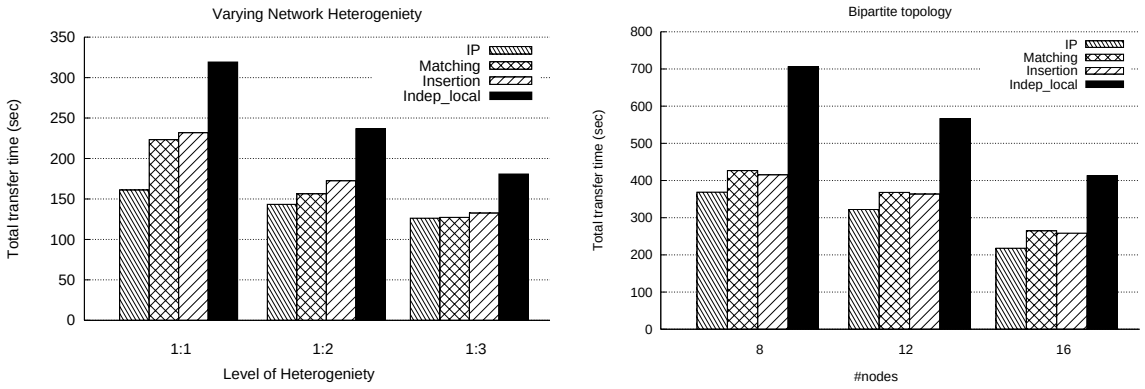


Figure 5.5: Performance of all schemes for a randomly generated workload

Fig 5.5 shows the comparison across the scheduling schemes in terms of the overall file transfer time(secs). These experiments were conducted using 4, 8, 12 compute nodes on randomly generated file request workloads by employing Mpich via tcp(ethernet). The initial distribution of files on the nodes was also chosen as random. The input request set consisted of around 50 file transfers each involving 1GB files. The results show that the IP scheme performs the best. This is because the IP formulation is able to integrate the global information of the file transfer request set and the platform topology information by incorporating it into its goal function of minimizing the overall file transfer time. It therefore, acts as a lower bound on the

overall file transfer time. The matching based approach *Matching* performs quite similar to the previously proposed insertion scheduling approach *Insertion*. This is because, the matching based approach leads to a contention minimizing schedule since the graph matching ensures that each at step, a set of non-conflicting file transfers are chosen to execute. *Indep_local* performs the worst as expected.



(a) (b)

Figure 5.6: (a) Performance of all schemes with varying network heterogeneity, (b) Performance of all schemes by employing a bipartite platform graph

Fig 5.6(a) shows the performance of the algorithms in terms of overall file transfer time (secs) on configurations with different degrees of network heterogeneity. This experiment was conducted over 12 nodes of the cluster by employing Mpich via tcp(ethernet). The workload used for this experiment is the same one corresponding to the results shown in Fig 5.5. Since the workload was random, each of the 12 nodes could possibly act as sources for some files and destinations for others. $(1:1)$ corresponds to the network homogeneous case while $(1:2)$ and $(1:3)$ correspond to network heterogeneity cases. We abstracted the platform graph as a fully-connected network and emulated heterogeneity by randomly choosing half of the links to have

double and triple the communication bandwidth as compared to the remaining links, respectively for the $(1 : 2)$ and the $(1 : 3)$ cases. The emulation is achieved by transferring proportionally smaller amounts of data on the faster links followed by locally padding the rest of bytes to the file. The results show that the performance gap between the IP approach and the other approaches decreases with increasing levels of network heterogeneity. At low heterogeneity, IP performs better because it explores a much larger search space of efficient solutions thereby achieving a better global solution. However, as the extent of heterogeneity increases, the search space of efficient solutions becomes more and more restricted to faster links and all the schemes take that into account.

Fig 5.6(b) shows the performance of the scheduling schemes on a bipartite topology platform graph. This experiment was conducted over 8, 12, 16 nodes of the cluster by employing the Infiniband interconnect. The bipartite topology was emulated by abstracting the topology as two distinct subsets of nodes with interconnection links only across the two sets. The workload for this experiment was a randomly chosen workload with the input request set consisting of multiple destination node mappings for each file. The size of each file in the workload was equal to 1GB. The result show expected trends except that the performance of $Indep\_Local$ is much worse than the other approaches. This is because each file needs to be sent to multiple different destinations, thereby leading to increased end-point contention due to multiple simultaneous requests for the same file.

Figure 5.7 shows the scalability results with varying number of compute nodes and varying number of input requests for IA. Since IP takes too long to execute even for moderately-sized workloads, therefore in this figures, we show results only

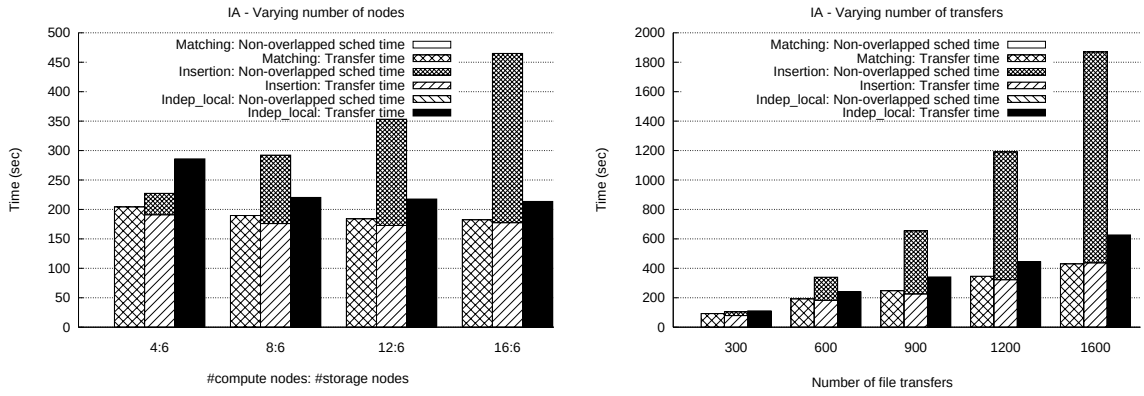


Figure 5.7: (a) Performance of different schemes for IA workload with varying number of nodes, (b) Performance of different schemes for IA workload with varying number of file transfers

for the other three schemes. To analyze the scalability of *Matching* with respect to the number of compute nodes, we ran experiments with an IA workload consisting of around 250 jobs over 4, 8, 12 compute nodes and 6 storage nodes. Note that the Figure 5.7 shows the performance in terms of two metrics, namely the total file transfer time and the non-overlapped scheduling time. The non-overlapped scheduling time is the difference between the end-to-end execution time and the total file transfer time, where end-to-end execution time is defined as the elapsed time between the instant when the scheduler accepts a batch of requests to the instant when all the requests have been finished. In other words, the non-overlapped scheduling time is the perceived scheduling overhead.

For *Insertion*, the end-to-end execution time is simply the sum of the scheduling time and the total file transfer time. This is because, for *Insertion*, the centralized scheduler generates the entire schedule once at the beginning followed by the transfer of files. However, *Matching* is a dynamic scheduling approach wherein the scheduler

generates the schedule in an iterative fashion while the file transfers are taking place. Therefore, the non-overlapped scheduling time is negligible and the end-to-end execution time closely matches the overall file transfer time. The base heuristic *Indep_local* also has negligible scheduling overhead. Figure 5.7(a) shows that *Matching* performs significantly better than *Insertion* in terms of the end-to-end execution time. This is because, non-overlapped scheduling time in *Matching* is very small. In terms of the total file transfer time, the performance of *Matching* is quite close to *Insertion*. Figure 5.7(b) shows the results with increasing number of requests for an IA workload. We observe that *Matching* is able to perform much better than *Insertion*. This is because *Insertion* has a quadratic dependence of its complexity on the number of requests as opposed to *Matching* which has a linear dependence.

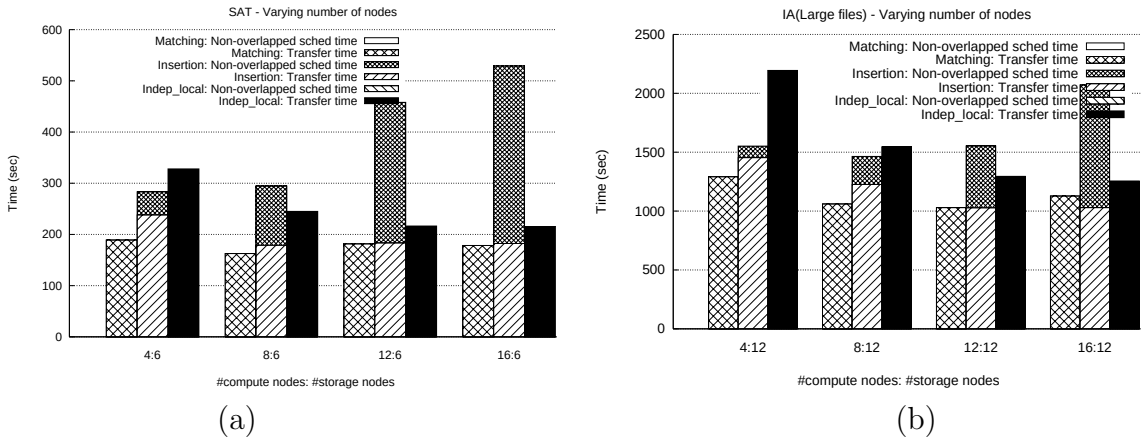
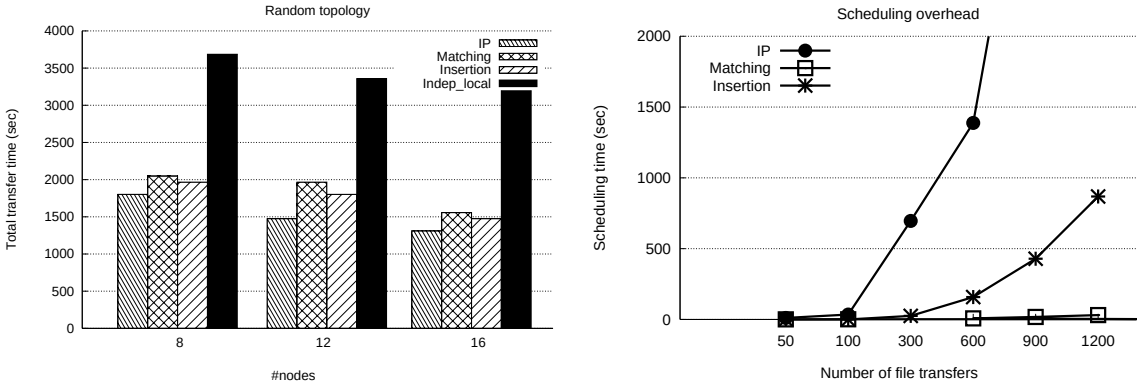


Figure 5.8: (a) Performance of different schemes for SAT workload with varying number of nodes, (b) Performance of different schemes for IA workload (large files)with varying number of nodes

Fig 5.8(a) shows the performance results for a SAT workload in terms of the total file transfer time and the non-overlapped scheduling time. We observe that

the *Matching* scheme outperforms *Insertion* by upon 20% in terms of the total file transfer time. In terms of the end-to-end execution time, *Matching* does significantly better than *Insertion*. Fig 5.8(b) shows the performance results for a larger IA workload involving transfer of 1GB and 200MB files initially distributed over 12 storage nodes. The number of compute nodes is varied from 4, 8, 12 to 16 nodes. The results show expected trends.



(a) (b)

Figure 5.9: (a) Performance of all schemes by employing a random platform graph, (b) Scheduling overhead for all schemes

Fig 5.9(a) shows the performance in terms of total file transfer time of the scheduling schemes on a random topology platform graph consisting of 8, 12 and 16 nodes respectively. These were obtained by using simulations involving transfer of 1GB files over 100Mbit/sec fast ethernet. The results show that IP approach performs the best as expected. The performance of the *Matching* heuristic is able to match the performance of the Insertion scheduling approach. Fig 5.9(b) shows the scheduling times for various schemes. The scheduling time shown is the actual time spent in generating the schedule. Essentially, it is the sum of the overlapped and the non-overlapped

scheduling times. IP has a high scheduling overhead for larger configurations, due to the exponential complexity of the search. The scheduling time of $Insertion$ is higher than that of $Matching$ for larger sized workloads, as expected.

5.5 Conclusion

We proposed two strategies for collectively scheduling a set of file transfer requests made by a batch of data-intensive jobs on heterogeneous systems - one approach formulates the problem using 0-1 Integer Programming and another based on using max-weighted graph matching. The performance results show that the IP formulation results in the best overall file transfer time. However, it suffers from high scheduling time. The graph matching based approach results in slightly higher file transfer completion times, but is much faster than the IP based approach. Moreover, the matching based approach is able to match the performance of the Insertion scheduling approach with a much lower scheduling overhead. Our conclusion is that the IP based approach is attractive for small workloads, while the matching based approach is preferable for large scale workloads.

In the next chapter, we look at the same problem as discussed in this chapter, but in a wide-area context. We propose efficient algorithms to schedule and execute the transfer of a set of files distributed across multiple machines to another set of machines in a wide-area environment.

CHAPTER 6

WIDE-AREA FILE TRANSFER SCHEDULING

In our proposed work so far, we looked at the problem of scheduling a batch of data-intensive jobs on homogeneous [51] clusters. We have also investigated effective use of file replication [50] and scheduling of file transfers in data center environments [47].

There are a number of projects that target shared access to distributed data in a wide-area environment. Biomedical Informatics Research Network (BIRN) [20] is an example of a project which targets shared access to distributed medical data in a grid environment. It focuses on collaborative access and analysis of distributed data which is generated by medical imaging studies. As another example, consider a multi-institutional study which collects and analyzes biomedical image data, obtained from high-resolution scanners to develop animal models of phenotype characteristics in disease progression. Hundreds or thousands of images can be obtained from a subject and there can be hundreds of subjects in a study. These images may be collected and stored at multiple sites. Researchers wishing to carry out an analysis using images from a large population of subjects will query image datasets at multiple sites. The image files extracted as a result of the query will then either be downloaded to a local system or be transferred to computational machines distributed in the environment

for processing. The GriPhyN project [39] aims to deliver data generated from large scale simulations and experimentation to physicists across a wide-area network. The TeraGrid [69] is a collaborative infrastructure which spans 8 sites across the US and is intended to facilitate distributed data-intensive scientific computing.

Consider the scenario where scientists at geographically distributed sites request data transfer to their local sites over the wide-area. In a wide-area environment, resource characteristics fluctuate and vary over time. Therefore, the ensuing data transfers may take a longer amount of time than expected due to congestion in the shared wide-area networks or due to temporary unavailability of replicas. In such a dynamic heterogeneous wide-area environment, there is need to develop scheduling mechanisms which are adaptive to resource performance fluctuations.

In this chapter, we propose efficient algorithms to schedule and execute the transfer of a set of files distributed across multiple machines to another set of machines in a wide-area environment. The objective is to minimize the total execution time of a batch of file transfer requests. A destination machine receives a subset of the files. The subsets of files assigned to different destination machines may overlap, i.e., a file may be mapped to multiple destination machine. Figure 6.1 shows that two different sources of a file $F1$ can be used simultaneously to transfer disjoint chunks of the file, thereby increasing the throughput. The figure shows that once a replica of $F1$ is created on the node $N1'$, then the node $N1'$ and the storage repository $D3$ can simultaneously transfer the file to the node $N1$.

We present a network flow based mixed integer programming (IP) formulation of the scheduling problem. The resulting solution is a lower bound on transfer time

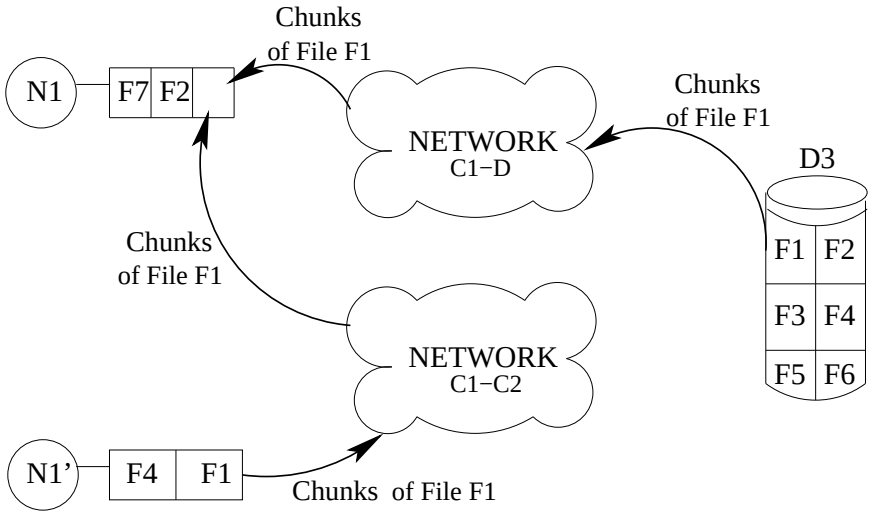


Figure 6.1: Simultaneous usage of multiple replicas of File F1

under idealistic conditions of resource availability and performance. We then propose a dynamic scheduling heuristic which employs network bandwidth information obtained from past GridFTP transfers to adapt its scheduling decisions, thereby, accounting for the resource availability fluctuations in the wide-area environment. The algorithm also employs adaptive replica selection, if files are replicated in the environment during previous transfers. It performs simultaneous transfer of portions of files from multiple replicas to maximize data transfer bandwidth. We have developed an implementation of our algorithm using GridFTP [11] as the underlying transport protocol for data transfers. We experimentally evaluate the algorithm on a wide-area network testbed consisting of clusters located at geographically disparate locations. The results show that the algorithm can take advantage of multiple replicas and concurrent data transfers.

6.1 Network Flow Formulation

In this section, we propose a mixed integer programming (IP) formulation of our target problem. The formulation is based on the maximization of network flows from sources to sinks. The wide-area environment is represented by a graph $G = (V, E)$, referred to here as the *platform graph*. In this graph, V is the set of machines and E represents the network edges. A network edge is the wide-area connection between two machines. The weight of the edge is a measure of the achievable bandwidth between the two machines. The set of two tuples $R = \{< f_\ell, v_d >\}$ represents that file f_ℓ needs to be transferred to the destination node v_d. The set of two tuples $D = \{< f_\ell, v_s >\}$ denotes that file f_ℓ is present on the source node v_s. Multiple replicas of a file may exist and each replica is represented by a two tuple in D.

The optimization problem solves for a set of variables $Flow_{ij\ell}$, where $Flow_{ij\ell}$ is the rate (bandwidth) at which file f_ℓ is transferred through the link between the nodes v_i and v_j.

Let $InFlow_{i\ell}$ be the rate at which the file f_ℓ enters the node v_i along the incoming edges.

$$(\forall \ell)(\forall i, i \in V) InFlow_{i\ell} = \sum_{(\forall j,(j,i)\in E)} Flow_{ji\ell} \tag{6.1}$$

For each file f_ℓ, the flow on each outgoing edge which emanates from the node v_i cannot exceed the inflow at which the file f_ℓ enters the node v_i. This necessarily holds true for all the nodes except the source node set for the file f_ℓ.

$$(\forall \ell)(\forall j)(\forall i, i \in V - \{k| < f_\ell, v_k > \in D\}) Flow_{ij\ell} \leq InFlow_{i\ell} \tag{6.2}$$

The total inflow rate of all the files entering a node v_i should not exceed the bandwidth capacity at the node v_i, $VCap(i)$. Similarly, the total outflow rate on all outgoing edges should not exceed the bandwidth capacity at the node v_i.

$$(\forall i, i \in V) \sum_{(\forall \ell)} InFlow_{i\ell} \leq VCap(i) \tag{6.3}$$

$$(\forall i, i \in V) \sum_{(\forall j)(\forall \ell)} Flow_{ij\ell} \leq VCap(i) \tag{6.4}$$

The aggregate flow rate for all the files through the edge e between the nodes v_i and v_j should not exceed the bandwidth capacity $ECap(ij)$ of the edge e.

$$(\forall i, i \in V)(\forall j, (i,j) \in E) \sum_{(\forall \ell)} Flow_{ij\ell} \leq ECap(ij) \tag{6.5}$$

We only consider destination nodes as intermediate nodes for other transfers. Therefore, an edge from a node v_i to a node v_j can have a non-zero flow for a file f_ℓ, only if the node v_j belongs to the destination node set for the file f_ℓ.

$$(\forall i, i \in V)(\forall j, (i,j) \in E)(\forall \ell, < f_\ell, v_j > \notin R) Flow_{ij\ell} = 0 \tag{6.6}$$

A feasible solution should not have flow cycles for each file f_ℓ. In other words, for each file f_ℓ, for all cycles in the graph G comprising only a subset of destination nodes for the file f_ℓ, the flow for the file on atleast one of the edges belonging to the cycle should be equal to zero. Let $Cycles_\ell$ be the set of all the cycles in the graph G consisting only of a subset of destination nodes of the file f_ℓ . Each element of the set $Cycles_\ell$ is a set of edges which constitute that cycle.

$$(\forall \ell)(\forall C, C \in Cycles_\ell)(\exists (i,j), (i,j) \in C) Flow_{ij\ell} = 0 \tag{6.7}$$

The finish time $FinishTime_{\ell k}$ of a transfer request for a file f_ℓ to its destination v_k is computed as follows.

$$FinishTime_{\ell k} = \frac{FileSize(\ell)}{InFlow_{k\ell}} \tag{6.8}$$

Note that the finish time is computed based on the total incoming flow to the destination node v_k for the file f_ℓ. We cannot use the total outgoing flow from the sources of the f_ℓ to compute the finish time since there are possibly multiple destinations for each file and therefore outgoing flow from a source node for a particular file is not necessarily the aggregate flow for a particular file-destination pair.

The objective is to minimize the total transfer time $Makespan = \max_{\forall \ell,k} FinishTime_{\ell k}$. Note the objective function is a non-linear function which means that the problem is non-linear optimization problem with linear constraints. We represent the objective function in an alternate way which makes the problem a linear optimization problem. We define a function $NormalizedRate_{\ell k}$ for each file transfer request.

$$NormalizedRate_{\ell k} = \frac{InFlow_{k\ell}}{FileSize(\ell)} \tag{6.9}$$

With the new formulation, the objective becomes the maximization of $MinRate$, which is the minimum value of $NormalizedRate_{\ell k}$ over all file transfer requests. The solution to this optimization problem will provide, for each file transfer request, the flow rates which the request should employ for each edge in the graph. These flow rates can then be used to find the total transfer time, $Makespan$. This value of total

transfer time acts as a lower bound under idealistic conditions of resource availability and performance.

The flow based IP formulation inherently assumes that all file transfers take place in parallel and simultaneous file transfers on the same link share bandwidth. From a theoretical standpoint, serializing transfers on a resource, or simultaneous execution with resource sharing, results in the same makespan. In practice, however, because of limited resources on nodes and the high cost of congestion on lossy wide-area links, which leads to a continuous loop between TCP slow-start and congestion-avoidance phases, the solution obtained by the IP cannot be achieved for large batches with thousands of requests. Moreover, the scheduling overhead of a mixed integer programming approach may be unacceptable, especially for large workloads and system configurations. Therefore, in our work, we employ the solution obtained by the IP as a lower bound on the total transfer time and use it as a yardstick to compare against our proposed dynamic scheduling heuristics which we discuss in detail in Section 6.2.

6.2 Dynamic Scheduling Algorithms

In our approach, scheduling is done per chunk basis. Chunk is a portion of the file being staged to a destination machine. Transfer of chunks for a file can be interleaved with transfer of chunks for other files. Our scheduling approaches are iterative, employ adaptive replica selection, and use of multiple sources for simultaneously transferring multiple pieces of the same file, i.e., non-overlapping portions of a chunk, *sub-chunks*, can be retrieved simultaneously from multiple file replicas. In a wide-area environment, the network is often the bottleneck. A good choice of replicas along with concurrent transfer of data can be expected to yield good performance. Thus, given

a graph G, the set of tuples $R = \{< f_\ell, v_d >\}$, the set of tuples $D = \{< f_\ell, v_s >\}$, our objective is then to compute a schedule that will minimize the total file transfer time. The schedule comprises of a set of four tuples $< V_s, v_d, c_\ell, t >$. Here, c_ℓ is a chunk of file f_ℓ to be transferred, v_d is the destination machine, V_s is the set of source machines, from which portions of the chunk c_ℓ will be transferred, and t is the time at which the transfer of the chunk will start.

Replica selection depends upon a number of factors like network bandwidths, round-trip times, and file sizes. Moreover, in a wide-area network, the network bandwidth may fluctuate considerably. In order to handle dynamic network characteristics, our approach carries out replica selection "at the level of chunks" in an adaptive manner. We should note that as files are staged to their respective destination nodes, these nodes can act as replica sources for other requests of the same file. We employ dynamic information obtained on the fly from previously executed file transfers to drive our scheduling and replica selection decisions.

In order to support adaptive replica selection at chunk level and concurrent use of multiple replicas, we redefine the request set R and the data structure D. We define the modified request set R' as the set of three tuples, $R' = \{< f_\ell, v_d,$ $cur\_request\_offset(\ell, d) > | < f_\ell, v_d > \in R\}$ denoting that f_ℓ needs to be transferred to the node v_d starting at the offset $cur\_request\_offset(\ell, d)$. This means that a subset of the file f_ℓ is already present at the node v_d up to an offset $cur\_request\_offset(\ell, d)$. The value of $cur\_request\_offset(\ell, d)$ will change with time as more and more chunks of file f_ℓ get written onto node v_d. The initial values of $cur\_request\_offset(\ell, d)$ are set to 0, since the transfer of a file will start at offset 0. Similarly, D is redefined as $D' = \{< f_\ell, v_s, last\_byte\_offset(\ell, s) > | < f_\ell, v_s > \in D\}$, representing that the file f_ℓ

is currently present on the node v_s up to the offset value $last\_byte\_offset(\ell, s)$. Here, v_s can be one of the original source nodes of the file, or a destination node, to which the file has already been partially transferred. In the case v_s is a destination node, the value of $last\_byte\_offset(\ell, s)$ will change over time as more and more chunks become available on v_s. The final value of $last\_byte\_offset(\ell, s)$ will be the size of the file, $size(f_\ell)$.

6.2.1 Global Dynamic Scheduling Algorithm

This scheduling scheme proceeds in steps and in each step it selects a pending file transfer request $< f_\ell, v_d, cur\_request\_offset(\ell, d) >$ from R' and computes a schedule for the request. A request is considered pending if the file associated with the request has not been completely transferred to its corresponding destination and no other chunk of this file is being transferred to the same destination. The schedule for a request consists of a four tuple with the following elements: (1) the set of replica locations (V_s) to be accessed to retrieve the data, (2) the size of the chunk ($ChunkSize$) which will be scheduled for transfer at the current scheduling instant, (3) the portions of the selected chunk to be obtained from each source, and (4) the TCP buffer sizes to be used for each connection.

In our current implementation, we employ GridFTP as the underlying transfer mechanism. Each source node runs a GridFTP server. Each destination node uses the GridFTP client side API to retrieve the portions of the file. Since a destination node can become a replica source for a file, a GridFTP server runs on each destination node as well. After the schedule for a chunk has been computed, the scheduler sends the schedule information to the corresponding destination node. The destination

node starts the retrieval of the chunk from the source nodes. The scheduler moves on to the next pending file transfer request and repeats the whole process. The overall scheduling scheme is illustrated in Algorithm 6.1.

Algorithm 6.1 Global Dynamic Scheduling Heuristic

Require: Platform $G = (V, E)$ and a set $R =$
$\{< f_\ell, v_d > \mid$ file f_ℓ is requested by node $v_d\}$
1: $R' = \{< f_\ell, v_d, 0 > \mid < f_\ell, v_d > \in R\}$
2: $D' = \{< f_\ell, v_s, size(f_\ell) > \mid < f_\ell, v_s > \in D\}$
3: $HostBw_i =$ the host bandwidth at node v_i
4: **while** there are pending requests, i.e., $R' \neq \emptyset$ **do**
5: **if** $\exists v_d$ such that $HostBw_d > \epsilon$ **then**
6: **for** each request r $= < f_\ell, v_d, cur\_request\_offset(\ell, d) > \in R'$ **do**
7: $< V_s, ChunkSize, TCPBufSize, SubChunkSize > \leftarrow$ $SelectReplicas(G, D', r)$
8: Compute the expected finish time to transfer the chunk of file f_ℓ to destination v_d.
9: Choose the request r with the minimum expected finish time
10: Schedule the transfer of the chunk of the file f_ℓ from replica nodes V_s to the node v_d.
11: $R' \leftarrow R' - \{r\}$
12: Update the expected available host bandwidth ($HostBw_i$) at the source and destination nodes.
13: **for** every completed chunk transfer $< V_s, v_d, c_\ell, t >$ **do**
14: Update the available network bandwidths between sources ($v_s \in V_s$) and node (v_d)
15: **if** $endOffset(c_\ell) < size(f_\ell)$ **then**
16: $R' \leftarrow R' \bigcup \{< f_\ell, v_d, last\_byte\_offset(endOffset(c_\ell), d) >\}$

At step 7, the replica selection method denoted as *SelectReplicas* is invoked to select replicas for the transfer request; the algorithm for replica selection is described in the next section. The output from this method makes up the schedule for the request. The next step (step 8) is to compute the expected minimum completion time for transferring a chunk of the requested file. The transfer completion time is

computed as follows. We first divide the aggregate chunk of size $ChunkSize$ into sub-chunks which will be fetched from each replica. The size of the sub-chunks are chosen to be in the same ratio as that of the bottleneck bandwidths between each source host and the destination. The transfer completion time is then simply the maximum of the times taken to send each sub-chunk from a source to the destination. At step 9, following the well-known MinMin [42] algorithm, among all the pending requests, the file transfer request with the minimum expected completion time is chosen to be scheduled on the set of resources which yield its minimum completion time. The overall process repeats until all the file transfers have been scheduled. The replication selection step, the determination of the chunk size, and dynamic bandwidth prediction are presented in detail in the following sections.

Replica Selection

The replica selection algorithm (Algorithm 6.2) proceeds as follows. For each replica location v_s, we record the bandwidth obtained through past GridFTP transfers to find the network bandwidths and end-to-end latencies from the location v_s to the destination v_d. To perform replica selection, we apply a two phase heuristic. Each phase involves applying a filtering condition to choose a subset of replica sources of the file to fetch the data. The first filtering condition is based on the file size and its relation to the slow start phase of TCP. The second filtering condition is based on the expected available bandwidth at the sources and destination of files as well as the expected available bandwidth in the network. The output of the second filtering condition is a subset of replicas to be used for transferring the file. The TCP buffer size and the size of the portion of the chunk to be fetched from each replica are also computed.

Algorithm 6.2 Replica Selection Algorithm

Require: A pending request $< f_\ell, v_d, off >$

1: **for** Each existing replica v_s of the file f_ℓ **do**
2: Compute the bandwidth delay product $BDP_s = NetBw_{s,d} \times RTT_{s,d}$ for the link between hosts v_s and v_d.
3: **if** $size(f_\ell) \geq C \times BDP_s$ **then**
4: add the replica v_s to the tentative replica set T_s
5: **for** each replica $v_s \in T_s$ in decreasing order of available bandwidth values to v_d **do**
6: Add the replica v_s to the final replica set V_s
7: Update the destination $HostBw_d$ to account for the transfer between v_s and v_d (if $NetBW_{s,d} > HostBw_d$ the transfer bandwidth between v_s and v_d will be $HostBw_d$)
8: **if** $HostBw_d < \epsilon$ **then**
9: **break**
10: **if** $V_s = \emptyset$ **then**
11: pick the source $v_s \in T_s$ with highest bandwidth and set $V_s \leftarrow \{v_s\}$
12: Compute $ChunkSize$, $TCPBufSize$, $SubChunkSize$ per replica, using V_s and v_d and available network bandwidth
13: **return** $< V_s, ChunkSize, TCPBufSize, SubChunkSize >$

TCP is a window-controlled transport protocol and the performance of a TCP connection is dependent on the Bandwidth-Delay product (BDP). The BDP of a network path is defined as the product of bandwidth Bw of the path and the round-trip time RTT. TCP has an initial slow start phase where in it gradually increases the send window size. If the TCP buffer size equals the BDP, the connection will be able to saturate the path, achieving the maximum possible throughput. However, if the amount of data to be transferred is lower than the BDP, the observed bandwidth will be smaller than the maximum achievable bandwidth. Hence, if the file size is smaller than a pre-determined multiple of BDP (step 3 in the algorithm), the replica is tentatively not considered for selection. Otherwise, the replica is added to the list of tentatively selected replicas T_s. The output of this phase yields a subset of replicas.

In the second phase, the subset of replicas, T_s, is further pruned based on the network bandwidth between each replica and the destination host and the bandwidth available at the destination hosts. Employing too many replica sources in parallel may overwhelm the destination host in which case each TCP connection may lose packets and hurt performance. Therefore, the replica sources should be chosen in a manner so that the aggregate in flow rate of packets matches the available bandwidth at the destination host. We use a greedy algorithm for selecting sources. For each replica location, a bottleneck bandwidth is computed as the minimum of the expected network bandwidth and the available bandwidth at the destination. We order the replica sources of the selected subset of replicas in non-increasing order of available bandwidth values to the destination node v_d, and choose them one by one until we saturate the bandwidth of the destination host.

If no replica is selected at the end of this phase, the best replica is chosen for the file and added to the set V_s (step 11). The best replica is simply the replica which yields the least completion time for the transfer and is chosen by taking into account the bandwidths from each replica location.

Chunk Size

The size of a chunk is decided statically. For a file transfer request, it is the maximum of a pre-determined fraction of the file size and a threshold value. The motivation behind this is the slow start and congestion control mechanism of TCP. If the size of the chunk on a certain network edge is less than the BDP, the transfer of the chunk will finish in the slow-start phase, thereby not permitting use of the maximum achievable bandwidth. The threshold value for a given file transfer request

is computed as a pre-determined multiple of the sum of the BDPs between each source replica and the destination node.

Dynamic Bandwidth Prediction

The bandwidth to access data from a file replica is an important factor in replica selection. Replicas with higher access bandwidth are expected to give better performance. The key issue is to determine an accurate measure of expected bandwidth from a replica. We employ bandwidth information obtained from previous GridFTP transfers to predict the future access bandwidths. For each file transfer that has finished so far, we track and save the information about the achieved bandwidth between the source-destination pair into a circular queue. We employ simple mean-based predictors to estimate the value of the bandwidth in the next interval. In future, we plan to employ more sophisticated techniques [75] for more accurate bandwidth predictions.

In addition, we employ a dynamic bandwidth scaling mechanism which works in a control feedback loop as follows. If the observed bandwidth between a given source-destination pair is able to meet a certain percentage of the expected bandwidth value for N successive transfers using the source-destination pair, the expected network bandwidth for the next file transfer between the two nodes is scaled up by a pre-determined constant, BW_SCALE. The new value of expected bandwidth is then used to calculate the TCP buffer size for the file transfer between the two nodes. However, If the observed bandwidth between a given source-destination pair is lower than a certain fraction of the expected bandwidth value for N successive transfers that use the source-destination pair, the expected network bandwidth for the next file transfer between those two nodes is scaled down by BW_SCALE.

6.2.2 Local Dynamic Scheduling Algorithm

The global dynamic scheduler presented in Section 6.2.1 coordinates all the data-transfers between multiple sources and destinations. In this section, we describe a simplified variant of the global dynamic scheduler, which only uses local information in each destination node. The key idea here is that clients act independently and there is no master which coordinates multi-site file transfers. Each client (destination node) makes requests for files it needs one by one irrespective of what other clients are doing. For each file transfer, a client employs dynamic bandwidth information obtained from past file transfers and uses multiple replicas to optimize the transfer time of each file transfer. In other words, the local scheduler employs optimizations to minimize the transfer time of each file in much the same way as the global dynamic scheduler. The difference is that the scheduling decisions is made by each destination node independently. The scheduling strategy is illustrated in Algorithm 6.3.

Algorithm 6.3 Local Dynamic Scheduling Heuristic

Require: Platform $G = (V, E)$ and a set $R = \{< f_\ell, v_d > \mid$ file f_ℓ is requested by destination $v_d\}$

1: On each destination node v_d independently **do**
2: **for each** file request $< f_\ell, v_d >$ in non-decreasing file size order **do**
3: **for each** chunk of file f_ℓ **do**
4: $< V_s, ChunkSize, TCPBufSize, SubChunkSize > \leftarrow SelectReplicas(G, D', r)$
5: Schedule concurrent transfer of the chunk of file f_ℓ from replica nodes V_s to node v_d.
6: When transfer completes, update the available bandwidths between sources ($v_s \in V_s$) and the destination node (v_d)

6.3 Experimental Results

We compare our dynamic scheduling approaches against the optimistic lower bounds we obtained via IP formulation and a baseline strategy, referred to here as `Naive Scheduling`. In the baseline strategy, each destination node picks a randomly chosen replica source for retrieving a file instead of employing dynamic bandwidth information or multiple replicas.

6.3.1 Experimental Setup

We employ GridFTP [11] as the file transfer protocol. GridFTP exposes a set of API calls [6] for setting the TCP buffer sizes and for obtaining portions of a file from a source. In our implementation, a master scheduler sends control information to clients (destination hosts). Each destination host calls $globus\_ftp\_client\_partial\_get()$ to inform a source of the file it needs along with the start and end offsets. This is followed by a series of asynchronous $globus\_ftp\_client\_register\_read()$ calls which are used to transfer data from the source.

The experiments were carried out across 4 clusters that are located at geographically distributed sites. The first site, the BMI cluster, is a memory/storage cluster at the Department of Biomedical Informatics at the Ohio State University. The cluster consists of 64 nodes with an aggregate 0.5 TBytes of physical memory and 48TB of disk storage. The second site, the CSE cluster, is a 64 node cluster located at the Department of Computer Science and Engineering at the Ohio State University. Each node of the cluster is equipped with two 3.6 GHz Intel processors and 2 GBytes main memory. The other two sites belong to the Teragrid [69] network. One of them is the ORNL NSTG cluster which consists of 28 dual processor 3.06 GHz Intel Xeon

	BMI	CSE	ORNL	ANL
BMI	880	880	100	4
CSE	880	880	120	4
ORNL	100	120	900	10
ANL	4	4	300	700

Table 6.1: Link bandwidths (Mbps) between a pair of nodes located at different sites.

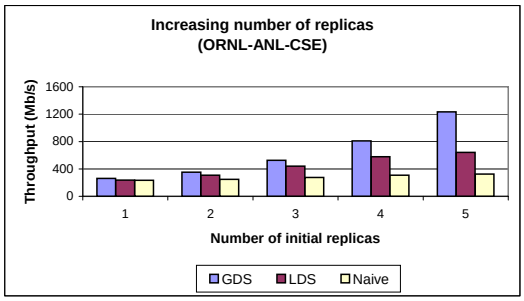
nodes. The other one is the UC/ANL IA-32 Linux cluster which consists of 96 dual-processor Intel Xeon nodes. Table 6.1 shows the bandwidths in Mbps(Megabits per second) between pair of nodes from different sites.

For evaluation, we compared the performance of the various scheduling schemes under a varying set of scenarios covering different file replica distributions, file-to-destination mappings and chunk sizes. For the experimental workloads, we employed three different file sizes corresponding to the files to be transferred. The sizes were 10MB, 50MB and 500MB respectively. In each workload, the fraction of the total number of files to be transferred for each file size was decided based on the distribution of these three file sizes in the GridFTP traces obtained from Globus metrics for a recent 12-month period [8]. The fraction of the number of files of each type is 0.5, 0.35 and 0.15 respectively for the 10MB, 50MB and 500MB files.

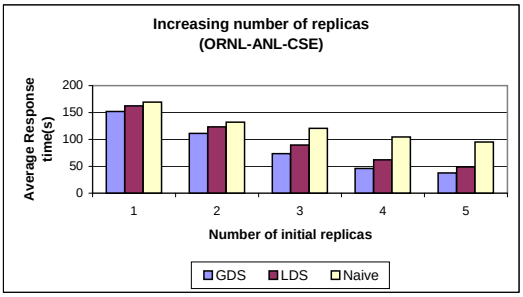
We measure the performance in terms of two metrics, namely, the average throughput which is the ratio of the total data transferred to the total execution time, and the average response time over all the requests in the workload.

6.3.2 Performance Evaluation

Figure 6.2 shows the relative performance of the Global Dynamic Scheduling (`GDS`), Local Dynamic Scheduling (`LDS`) and `Naive Scheduling` schemes on workloads with

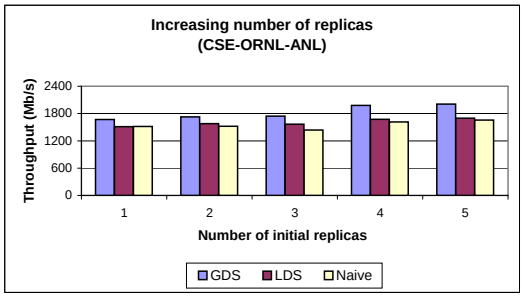


(a)

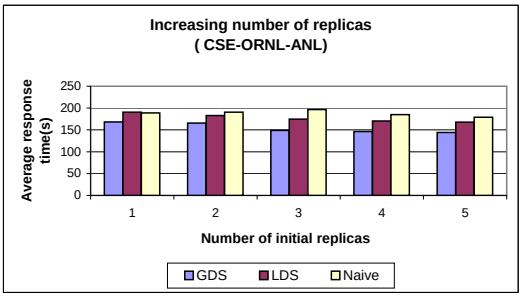


(b)

Figure 6.2: Performance of all the algorithms with increasing number of replicas (replicas added in the order ORNL-ANL-CSE) in terms of the (a) Average throughput and (b) Average response time.

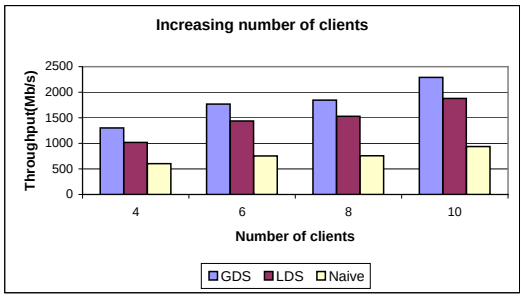


(a)

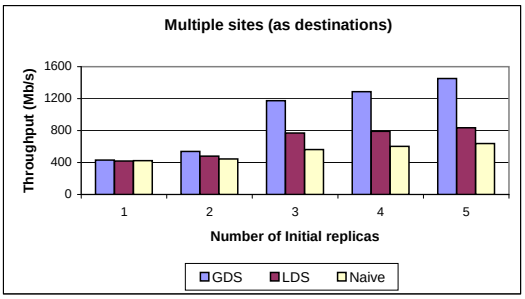


(b)

Figure 6.3: Performance of all the algorithms with increasing number of replicas (replicas added in the order CSE-ORNL-ANL) in terms of the (a) Average throughput and (b) Average response time.



(a)



(b)

Figure 6.4: (a) Performance in terms of throughput (Mbps) of all the algorithms with increasing number of clients (b) Performance in terms of throughput (Mbps) of all the algorithms for workloads where multiple sites act as clients as well as sources.

increasing degree of replication. This experiment was conducted across the 4 sites (BMI-ORNL-ANL-CSE) in a (4-3-2-3) configuration. The numbers in the parentheses refer to the number of nodes employed at each site, respectively. The input request set consisted of 300 files, the size of each of which is one of the three aforementioned values. In addition, the request set consisted of multiple destination node mappings for each file. In this experiment, all the requests in the input set had their destination as one of the nodes of the BMI cluster. The degree of replication here refers to the average number of file replicas present in the environment. Initially, the replicas were placed only on the ORNL nodes (average number of initial replicas being 1 or 2). Then, the degree of replication is increased by placing more file replicas on the ANL and CSE nodes. For the cases where the average number of replicas is one or two, all the file transfers employ either the ORNL cluster (initial replicas) or the BMI cluster (as files are created on the BMI nodes, they themselves can act as file replicas). The node-to-node bandwidth from an ORNL node to a BMI node is around 100Mbps. However, the bandwidth for a send from an CSE node to a BMI node is around 880Mbps. Therefore, as the degree of replication increases, the average throughput shows a significant performance improvement for the `GDS` scheduler. This is because, as replicas are placed on the CSE cluster, the algorithm makes an intelligent choice of choosing the CSE replicas more often than the other replicas. The results also show that the `GDS` is able to consistently outperform the other two approaches. In the other two schemes, clients act independently and make requests for files without any coordination. Each file needs to be sent to multiple different destinations, leading to increased end-point contention due to multiple simultaneous requests for the same file. Therefore, the performance improvement in these schemes due to increased replication

is offset by the endpoint contention caused due to uncoordinated local scheduling. In terms of the average response time, `GDS` performs the best. `GDS` schedules the requests with the minimum expected completion time first. On the other hand, in `LDS` and `Naive Scheduling`, since multiple clients act independently of each other, requests with higher expected completion times can possibly execute before requests with lower expected completion times, thus increasing the overall response time.

Figure 6.3 shows the relative performance of the various scheduling schemes with increasing degree of replication. However, in this case, the initial replication is handled differently. The replicas were initially placed only on the CSE nodes. The degree of replication is then increased by placing more file replicas on the ORNL and ANL nodes respectively. This experiment was conducted by employing the same system configuration employed in the experiment corresponding to Figure 6.2. The input request set consisted of 1500 file transfers. The results show that as the number of replicas increase, the average throughput does not show a significant increase, as expected. More and more replicas were placed on nodes with low link bandwidths to the destination, resulting in no significant performance improvement.

Figure 6.4(a) shows the the relative performance of the various scheduling schemes on workloads with increasing number of clients (destination hosts). This experiment was conducted across the 4 sites (BMI-ORNL-CSE-ANL) in a (10-3-3-2) configuration. The numbers in the parentheses are the number of nodes employed at BMI, ORNL, CSE, and ANL, respectively. During the experiment, the number of nodes on the BMI cluster was varied from 4 to 10. Each request in the input set was destined to one of the BMI nodes. The number of requests in the input set varied from 300

for the 4 BMI nodes to around 600 file transfers for the 10 BMI nodes. Again, the request set consisted of multiple destination node mappings for each file. The degree of replication in these experiments refers to the average number of file replicas initially present. The average number of initial file replicas was set to 5. The figure shows that as the number of clients increase, the throughput increases. This is because, as file replicas are created on BMI nodes, these replicas also act as sources for other requested transfers of the same file. Even though the aggregate amount of transferred data increases as the number of BMI clients increases, the aggregate bandwidth increases by a greater factor, leading to increased throughput. Furthermore, the extent of performance improvement is maximum for the `GDS` scheduler. As the number of clients increase, the effects of end-point contention is expectedly higher. `GDS` makes a better job of accounting for contention by making efficient coordinated scheduling decisions, whereas the other schemes make client-side local decisions which cause a lot of end-point contention.

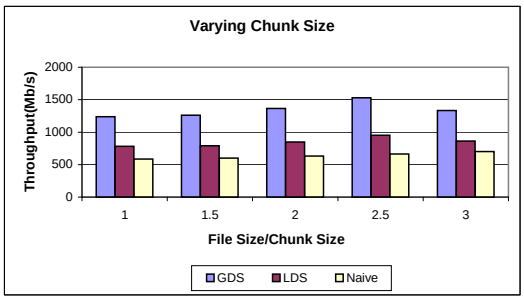


Figure 6.5: Performance of all the algorithms with decreasing chunk size.

Figure 6.4(b) shows the the relative performance of the various scheduling schemes on workloads with nodes belonging to different sites acting as destinations. This

N	CSE-ORNL-ANL (Single dest.)			ORNL-ANL-CSE (Single dest.)		
	Lower bound	GDS	% Increase	Lower bound	GDS	% Increase
1	148.6	201.4	36	250	289.3	16
2	142.6	193.5	36	163.2	195.9	20
3	134.6	191.6	42	125.4	165.65	32
4	134.6	183.4	36	114.2	149.7	31
5	134.6	157.8	17	79.4	125.12	58

Table 6.2: Comparison (in terms of transfer time (seconds)) between lower bounds and *GDS* scheduling algorithm for single-destination workloads. Here N represents the average number of initial file replicas.

experiment was conducted across the 4 sites (BMI-ORNL-CSE-ANL) in a (4-3-3-2) configuration. The input request set consisted of around 450 file transfers, the destination nodes for each file transfer were evenly distributed across the BMI, ORNL and CSE nodes. In this case, initially, the replicas were placed only on the ORNL nodes. Then, the degree of replication is increased by placing more file replicas on the ANL and CSE nodes. As is seen from the figure, as the number of replicas is increased, the performance gap between `GDS` and the other algorithms increases. An increase in the number of replicas (with replicas being added to the CSE cluster) creates more opportunity for faster transfers and more parallelism. `LDS` and `Naive Scheduling`, however, experience a lot of end-point contention, since each node can possible act as a source and a destination for multiple files simultaneously.

In the experimental results shown so far, the replica selection algorithm only chose fully-written files as replica sources for getting portions of files. In other words, a file which is in the process of being written to a destination node v_d cannot act as a source for the transfer of the same file to another node $v_{d'}$, even if the required portion of the file at the destination $v_{d'}$ has already been written at node v_d. Once

the file is completely written at node v_d, it can act as a replica source for other file transfers. We relaxed this constraint to allow for chunk-level replica sources. That is, a file which has not been completely written to a node v_d can still act as a source for other transfers of the same file. Figure 6.5 shows the performance results as a function of decreasing chunk size for all three algorithms by incorporating chunk-level replica sources. Here, the x-axis denotes the fraction $\frac{FileSize}{ChunkSize}$. Increasing the value of this parameter implies the file is transferred in smaller and smaller chunks. The results show that the throughput increases by employing smaller chunks up to a certain point, after which it shows a decrease. The initial increase is due to the fact that as the chunk size decreases, the number of possible replica sources for each file increases. Since each file has multiple destinations, chunks of file being written to some destination nodes can act as sources for other destination nodes. However, as the chunk size decreases further, the latency and I/O overheads of transferring the file in a greater number of chunks offset the potential benefit due to an increased number of file replicas.

6.3.3 Lower-bound Comparisons

Tables 6.2 and 6.3 show the comparison of the lower bounds obtained from the IP formulation in Section 6.1 with the experimental values obtained by employing the *GDS* scheduling algorithm for single-destination workloads and multiple-destination workloads. A single-destination workload, here, refers to a workload where each file has a single destination. A multi-destination workload, on the other hand, is one in which each file is transferred to multiple destination nodes. The multi-destination workloads employed here are the same as the ones which have been used for the

N	CSE-ORNL-ANL (Multiple dest.)			ORNL-ANL-CSE (Multiple dest.)		
	Lower bound	GDS	% Increase	Lower bound	GDS	% Increase
1	347.6	611.6	76	508.8	783.44	54
2	323.5	591.5	83	295.3	580.4	96
3	322.6	585.08	81	196.3	387.7	97
4	307.6	515.8	68	116.5	252.3	117
5	307.6	508.01	65	81.5	165.33	103

Table 6.3: Comparison (in terms of transfer time (seconds)) between lower bounds and GDS scheduling algorithm for multi-destination workloads. Here N represents the average number of initial file replicas.

results shown in Figures 6.2 and 6.3. The lower bounds have been computed by employing peak values of bandwidth on the various network links. The results show that the GDS scheduling algorithm results in between 16-58% increase in execution time compared to the lower bound for the single-destination workloads and between 54-117% increase for the multiple-destination workloads. The difference between the lower bound and the GDS is attributed to the fact that observable network bandwidth over the wide-area can show fluctuations over time. Also, because of the slow-start mechanism of TCP, some file transfers cannot observe the achievable network bandwidth. The difference between the lower bound and GDS is higher in the multi-destination case as compared to the single-destination case. The IP formulation can yield solutions which employ multiple destinations v_1, v_2, $...v_{k-1}$ of a file to send flow to another destination v_k. However, the formulation does not capture if the sources v_1, v_2, $...v_{k-1}$ have the required chunks of files or not at a given instant. In the worst case, all the sources might have the same chunks of file at a given instant, which means that only a single source would be used to transfer the chunk. The IP formulation is oblivious to this since it is based on static flow concepts and

does not incorporate the notion of time. Therefore, it results in an overestimation of the achievable throughput and a lower transfer time. Note that since the GridFTP servers do not have the capability to route the incoming data to a different GridFTP server, we do not allow this in the IP formulation as well. Using a GridFTP server as an intermediate node without storing the data on to the disk is non-trivial and require changes/additions to the GridFTP code and is a part of our future work.

6.3.4 Scheduling overhead

In our system, the scheduler computes the schedule information for a chunk request and sends this information to the corresponding destination node. The destination node starts the retrieval of the chunk from the source nodes. The scheduler moves on to the next pending file transfer request and repeats the whole process. Therefore, the scheduling performed by the centralized master and the file transfers between slave nodes occur in parallel. The end-to-end execution time is defined as the elapsed time between the instant when the scheduler accepts a batch of requests to the instant when all the requests have been completed. The non-overlapped scheduling time is the difference between the end-to-end execution time and the total file transfer time. In other words, the non-overlapped scheduling time is the perceived scheduling overhead. In our experiments, we observed that the non-overlapped scheduling time is negligible. This is because, the schedule is generated iteratively while the file transfers are taking place.

6.4 Conclusion

We presented a dynamic scheduling algorithm which schedules a set of data transfer requests made by a batch of data-intensive jobs in a wide-area environment like

the Grid. It also proposes a network flow based integer programming formulation of the scheduling problem, which is used to find a lower bound on the transfer time under idealistic conditions of resource availability and performance. The proposed dynamic scheduling algorithm is adaptive in that it accounts for network bandwidth fluctuations in the wide-area environment. The algorithm incorporates simultaneous transfer of disjoint chunks of the same file from different replica sources to a destination node, thereby increasing the aggregate bandwidth. Adaptive replica selection is used for transferring different chunks of the same file by taking dynamic network information into account. We employ GridFTP for data transfers and utilize information from past GridFTP transfers to perform predictive bandwidth estimations. We have shown the effectiveness of our algorithm through experimental evaluations on a wide-area testbed. In general, the proposed algorithm is expected to provide greater benefits with increasing degree of data replication. With a very low degree of replication, the algorithm is restricted in its choice of multiple replicas, thereby not giving significant performance improvements. Moreover, it is expected to perform well for multi-destination workloads. This is because, it, can dynamically take into account the existence of new replicas as some of the files are transferred to their respective destination nodes, and employ those replicas for subsequent file transfers.

In the next chapter, we extend the collective file-transfer scheduling heuristic proposed in this chapter, by incorporating the effects of multi-hop path splitting and multi-pathing to improve the file transfer performance in GridFTP.

CHAPTER 7

MULTI-HOP PATH SPLITTING AND MULTI-PATHING OPTIMIZATIONS FOR WIDE-AREA DATA TRANSFERS

High-bandwidth, high-latency optical networks are being increasingly used by researchers and scientists. These networks enable the transfer of extremely large files with sizes up to a few petabytes. A file transfer mechanism which can optimize the overlay routes used to transfer files and take advantage of the available network parallelism can enhance the data-transfer throughput achieved by an application. In addition, a lot of scientific experiments may involve the transfer of data over public, shared networks, instead of a dedicated network infrastructure. Here it is important for the file transfer mechanism to make intelligent use of available paths to maximize the achievable bandwidth.

GridFTP [10] is a widely used protocol which enables secure, reliable and high performance data movement. It facilitates efficient data transfer between end-systems by employing techniques like multiple TCP streams per transfer, striped transfers from a set of hosts to another set of hosts, and partial file transfers. By default, GridFTP employs TCP as the underlying transport protocol. Multiple TCP streams can be created between the source and the destination in order to offset the network

congestion and improve throughput. The use of multiple streams in parallel, however, does not affect the routing or take into account network parallelism.

In this chapter, we explore the effects of multi-hop path splitting and multi-pathing to improve the file transfer performance in GridFTP. Multi-hop path splitting improves performance by replacing a direct TCP connection between the source and destination by a multi-hop chain through some intermediate nodes. Multi-pathing involves striping the data at the source and sending it across multiple overlay paths thereby leading to a better achievable throughput. In other words, multiple independent routes can be employed to simultaneously transfer disjoint chunks of a file to its destination. We propose a path determination heuristic which incorporates these optimizations for efficient transfer of a single file. To optimize performance for batch file transfer requests, we extend the collective file-transfer scheduling heuristic discussed in Chapter 6. The extended algorithm incorporates multi-hop path splitting and multi-pathing optimizations.

We experimentally evaluate the optimizations and their GridFTP implementation on a wide-area testbed. Our performance metric in this work is the total transfer time of a batch of file transfer requests. We investigate performance improvements under several different file transfer patterns: one-to-all broadcast, all-to-one gather and data redistribution patterns common in many application scenarios. As an example, a one-to-all communication pattern may occur in high-energy physics, where the data stored at a US Tier-1 site needs to be broadcast to all Tier-2 sites. As another example, an all-to-one gather operation may arise when a researcher runs a biomedical image analysis on a local machine using a subset of images collected at different sites. Our results show that the proposed optimizations lead to significant

performance improvements for communication patterns like one-to-all, gather, and data redistribution from a set of source nodes to a set of destination nodes. However, we also observe that the improvements are significantly lower when data replication is employed.

7.1 Data Transport Optimizations

In this section, we provide an overview of the optimizations investigated in this work.

7.1.1 Multi-Hop Path Splitting

The observed throughput of GridFTP transfers in a wide-area environment may be lower than the maximum achievable throughput, due to a number of factors, such as the slow-start and congestion control mechanisms used by TCP. The technique of dividing a TCP connection into a set of shorter, better performing connections by splitting it at multiple intermediate points with the goal of improving the overall throughput has been studied [13, 16, 22, 66]. A split-TCP connection may perform better than a single end-to-end TCP connection due to several reasons. First, the round-trip time on each intermediate hop is shorter as compared to the direct end-to-end path. The congestion control mechanism of TCP would sense the maximum throughput quickly thereby attaining steady state, wherein it will give maximal possible throughput until a congestion event occurs. Second, any packet loss is not propagated all the way back to the source but only to the previous intermediate hop. In this work, as an alternative to a direct TCP connection between a source and destination, we explore the use of multi-hop pipelined transfers using intermediate nodes. If the bandwidth on each of the intermediate hops is higher than the direct

path, the overall throughput can be expected to improve. Figure 7.1 illustrates the use of multi-hop paths to transfer a file from a source to a destination. The direct path from the node C1 to C4 has the bandwidth B_{14} while an alternate path from the node C1 to C4 comprises of three hops, C1-C2, C2-C3 and C3-C4. The bandwidth on the multi-hop path B_{split} equals the minimum of the bandwidth on each of the hops.

$$B_{split} = \min\left(B_{12}, B_{23}, B_{34}\right) \tag{7.1}$$

The multi-hop path is preferable for transfer of data from node C1 to C4 if the B_{split} is greater than B_{14} and the end-to-end latencies of the multi-hop path and the direct path are comparable. We also refer to the alternate multi-hop path as a "split" path.

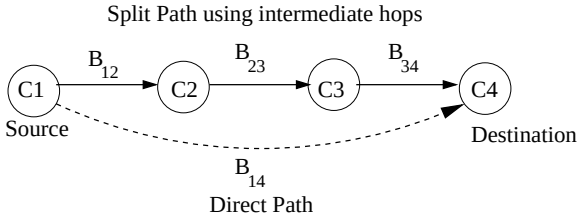


Figure 7.1: A multi-hop path C1-C2-C3-C4 can be used to transfer a file from C1 to C4 in a pipelined fashion.

7.1.2 Multi-Pathing

Striping the data at the source and sending it across multiple overlay paths can also lead to a better achievable throughput. In other words, multiple independent routes can be employed to simultaneously transfer disjoint chunks of a file to its destination.

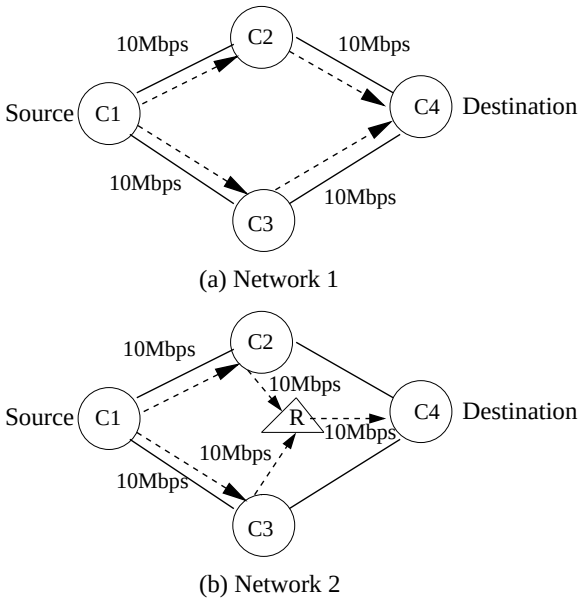


Figure 7.2: (a) Network 1 with two independent paths between C1 and C4. (b) Network 2 where the paths between C1 and C4 share a common bottleneck.

Splitting a file at the source and transferring it through multiple independent routes is equivalent to solving the maximum flow problem in a graph, where the graph vertices represent the overlay nodes (e.g., GridFTP servers), and the edge capacities equate to the network bandwidth between the corresponding pair of nodes. However, a direct application of maximum flow concepts does not account for bottlenecks due to physical sharing of links and routers. For example, in Figure 7.2(a), two independent paths exist between C1 and C4. Therefore, the overall bandwidth for file transfers between C1 and C4 can be increased by utilizing the two paths simultaneously provided the hosts C1 and C4 can handle the combined data rate. However, if there is a shared link which becomes bottleneck, the overall bandwidth would not increase by increasing the number of overlay transfers. In Figure 7.2(b) an additional routing node, R, is present, which is not a part of the overlay network. The routes C1-C2-C4 and C1-C3-C4 both share the bottleneck link R-C4. Therefore,

an approach that finds multiple parallel links, but does not consider the physical "underlay" network, will find suboptimal solutions. This is key to maximizing the effective aggregate bandwidth.

7.1.3 Path Determination Algorithm

The path determination algorithm (Algorithm 7.1) is an iterative algorithm that computes a set of paths which can be collectively used to transfer a file from its source node(s) to its destination node. At each step, algorithm invokes Best Path heuristic (Algorithm 7.2) to find a path that will yield minimum transfer time for the requested file given the concurrent transfers in the overlay network. It then modifies the overlay network graph to reflect the current transfer, and continues it search for another path. Since past research has shown that most of the benefit that can be obtained by splitting TCP can be achieved by using up to 2 hops and adding extra hops does not yield significant benefit [64], in this algorithm, we only consider paths of length 2 or 3 when looking at multi-hop paths.

In this work, the wide-area environment is represented by a graph $G = (V, E)$, referred as the *platform graph*. In the platform graph, V is the set of machines and E represents the network edges. A network edge is the wide-area connection between two machines. The weight of the edge w_{ij} is a measure of the achievable bandwidth BW_{ij} between the two machines. Let RTT_{ij} be the round-trip time of the wide-area path between the nodes v_i and v_j. The best path heuristic takes as input the overlay Graph $G = (V, E)$, the set of the sources V_s of the file f_ℓ, and the destination node v_d.

The best path to transfer a file from the source or a set of sources to a destination, is the path which yields the minimum expected completion time to transfer the file. The best path heuristic is a variant of the Dijsktra's shortest path algorithm. The algorithm involves creating a new Graph $G' = (V', E')$ where $V' = V$ and $E' = E$. However, the weighting function employed for weight assignment to an edge between the nodes v_i and the node v_j is the the ratio of the round-trip time of the path corresponding to the edge to its bandwidth. The motivation behind this is to give preference to low-latency high-bandwidth edges. The other difference is the calculation of the distance function to measure the goodness of a path. Since the transfer of a file from a source node to a destination node through a multi-hop path occurs in a pipelined fashion, therefore, the distance function of a path is computed as the maximum of the weights on each of its constituent edges (see step 18). Note that the traditional shortest path algorithm employs the distance function to be the sum of weights instead of the maximum.

Algorithm 7.1 Path Determination

Require: Platform $G = (V, E)$, request $< f_\ell, v_d >$, where the file f_ℓ is requested by destination v_d and a set V_s representing the set of sources of the file f_ℓ.

1: $Chosen\ Paths = \emptyset$
2: **while** 1 **do**
3: Run the Best Path Heuristic. Let $MinPath$ be the output.
4: Let $MinWeight$ be the minimum weight edge on the path $MinPath$.
5: **if** $MinWeight == 0$ **then**
6: return
7: **else**
8: Add the path $MinPath$ to the set $Chosen\ Paths$.
9: **for** Each edge $(v_i, v_j) \in MinPath$ **do**
10: $w_{ij} = w_{ij} - MinWeight$
11: **return** $Chosen\ Paths$

Algorithm 7.2 Best Path

Require: Platform $G = (V, E)$, request $< f_\ell, v_d >$, where the file f_ℓ is requested by destination v_d and a set V_s representing the set of sources of the file f_ℓ.

1: $Output\ Path = \emptyset$
2: Create a new graph $G' = (V', E')$ with $V' = V$ and $E' = E$. Weight of an edge w_{ij} in the Graph G' equals $\frac{RTT_{ij}}{NetBW_{ij}}$.
3: Define a variable $Dist_i$ for each vertex.
4: Define a variable $HopCount_i$ for each vertex.
5: Define a variable $Pred_i$ for each vertex.
6: **for** each node $v_i \in V'$ **do**
7: **if** $(v_i \in V_s)$ **then**
8: $Dist_i = 0$
9: **else**
10: $Dist_i = \infty$
11: **for** each node $v_i \in V'$ **do**
12: $HopCount_i = 0$
13: $Pred_i$ = -1
14: Unmark all vertices of the Graph G'
15: **while** There exists an unmarked vertex in G' **do**
16: Pick the unmarked vertex v_i with the minimum value of $Dist_i$ among all unmarked vertices.
17: **for** Each vertex v_j adjacent to v_i **do**
18: **if** $(Dist_j \geq \max(Dist_i, w_{ij})) \wedge ((HopCount_i \leq 2 \wedge v_j == v_d) \vee (HopCount_i < 2 \wedge v_j \neq v_d))$ **then**
19: $Dist_j = \max(Dist_i, w_{ij})$
20: $Pred_j = i$
21: $HopCount_j = HopCount_i + 1$
22: Set the vertex v_i as marked.
23: $Output\ Path = v_d$
24: $split = Pred_d$
25: **while** $split \neq -1$ **do**
26: Prepend $split$ to $Output\ Path$
27: $split = Pred_{split}$
28: **return** $Output\ Path$

7.1.4 Modeling Bottleneck due to Shared Resources

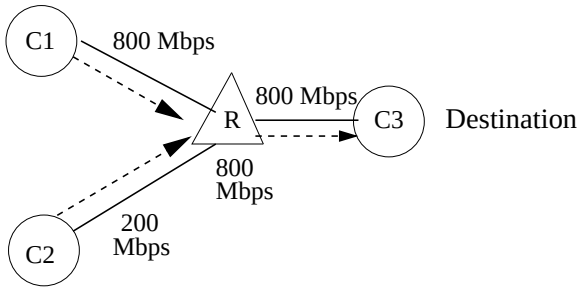


Figure 7.3: An example setup with shared resources.

The path determination algorithm presented in Section 7.1.3 chooses a set of independent paths to collectively transfer a file. However, one or more selected paths can possibly share bottleneck links, which means that the overall bandwidth would not necessarily increase by employing multiple paths. In some cases, the aggregate bandwidth might sometimes even decrease by employing multiple paths. An example of a setting with shared routers and links is illustrated in Figure 7.3. In this setting, two paths, C1-R-C3 and C2-R-C3, can be simultaneously utilized to transfer data. If the existence of router R is oblivious to the multi-pathing decision algorithm, then it will choose to split a file of size say 1000 Mb into two parts, one of size 800 Mb which is transferred along the path C1-R-C3, and the other of size 200 Mb which is transferred along the path C2-R-C3. Since the router R can only sustain a bandwidth of 800Mbps, the flow along the path C1-R-C3 will saturate R. In that case, the two flows are effectively serialized, requiring 2 seconds to transfer the file. The aggregate bandwidth, therefore, is 500Mbps. On the other hand, a multi-pathing decision, which incorporates the knowledge of the existence of R, can choose to send the entire flow along the path C1-R-C3, thereby getting a throughput of 800Mbps.

We model the shared bottlenecks by performing an offline characterization of the network. For each pair of edges $< e_1, e_2 >$ in the graph G, we first measure the end-to-end throughput for the wide-area paths corresponding to edges e_1 and e_2 with no interference. Then we measure the end-to-end throughput on the two edges by performing file transfers along them in parallel. The measured throughput values along e_1 and e_2 are then used to figure out if the two edges share a common bottleneck. The output generated by this analysis is a set of two-tuples $Shared$, where each element of the set $Shared$ represents a set of edges in the overlay which share a common bottleneck. For example, if $Shared = \{< e_1, e_2 >, < e_3, e_4 >\}$, it means that edges e_1 and e_2 share a common bottleneck and so do the edges e_3 and e_4. We ran the traceroute utility on the elements of the set $Shared$ and verified that the edges indeed involved shared links.

The offline characterization corresponding to identification of shared bottlenecks is then used to avoid choosing multiple overlay paths wherein the aggregate bandwidth would not increase. Algorithm 7.3 shows a variant of the proposed Path Determination algorithm which incorporates this knowledge.

7.1.5 Scheduling a Batch of File Transfers

The optimizations presented in Sections 7.1.1 – 7.1.4 aim to improve performance for single-file transfers. In some applications, a batch of file transfers need to be handled. In chapter 6, we proposed a dynamic scheduling algorithm which schedules a set of file transfer requests made by a batch of data-intensive jobs in a wide-area environment. The scheduling algorithm is iterative, employs adaptive replica selection, and makes use of multiple sources for simultaneously transferring multiple pieces of

Algorithm 7.3 Path Determination with modeling of shared bottleneck

Require: Platform $G = (V, E)$, request $< f_\ell, v_d >$, where the file f_ℓ is requested by destination v_d and a set V_s representing the set of sources of the file f_ℓ.

```
1: Chosen Paths = ∅
2: while 1 do
3:    Run the Best Path Heuristic. Let the path returned be represented by MinPath.
4:    Let MinWeight be the minimum weight edge on the path MinPath.
5:    if MinWeight == 0 then
6:       return
7:    else
8:       for Each Path APath ∈ Chosen Paths do
9:          if APath and MinPath share an underlying bottleneck then
10:            return
11:      Add the path MinPath to the set of selected paths Chosen Paths.
12:      for Each edge (v_i, v_j) ∈ MinPath do
13:         w_ij = w_ij − MinWeight
14: return Chosen Paths
```

the same file, i.e., non-overlapping portions of a chunk, *sub-chunks*, can be retrieved simultaneously from multiple file replicas. The scheduling scheme, however, did not incorporate multi-hop path splitting or multi-pathing. The path to transfer a file from a source node to a destination node follows the underlying network routing. In this chapter, we propose a new dynamic scheduling algorithm which builds upon the algorithm proposed in our previous work by incorporating the ideas of multi-hop path splitting and multi-pathing.

This scheduling scheme proceeds in steps and in each step it selects a pending file transfer request $< f_\ell, v_d >$ from the request list, R, and computes a schedule for the request. The schedule for a request consists of the set of paths to be employed to collectively transfer the file.

In our current implementation, we employ Globus GridFTP as the underlying transfer mechanism [11]. Each source node runs a Globus GridFTP server. Each destination node uses the GridFTP client side API to retrieve the portions of the file. Since a destination node can become a replica source for a file, a GridFTP server runs on each destination node as well. After the schedule for a file has been computed, the scheduler sends the schedule information to the corresponding destination node. The destination node starts the retrieval of the file from the source nodes. The scheduler moves on to the next pending file transfer request and repeats the whole process. The overall scheduling scheme is illustrated in Algorithm 7.4.

Algorithm 7.4 Global Dynamic Scheduling with Multi-Hop Path Splitting and Multi-Pathing

Require: Platform $G = (V, E)$ and a set $R = \{< f_\ell, v_d > \mid$ file f_ℓ is requested by destination $v_d\}$

1: $HostBw_i =$ the host bandwidth at node v_i
2: **while** there are pending requests, i.e., $R \neq \emptyset$ **do**
3: **if** $\exists v_d$ such that $HostBw_d > \epsilon$ **then**
4: **for** each request $r =< f_\ell, v_d > \in R$ **do**
5: Create a new graph G' identical to G.
6: $ChosenPaths \leftarrow$ Path Determination(G', r)
7: Compute the expected finish time to transfer the file f_ℓ to destination v_d.
8: Choose the request r with the minimum expected finish time
9: Schedule the transfer of the file along the selected paths to the node v_d.
10: $R \leftarrow R - \{r\}$
11: Update the expected available host bandwidth ($HostBw_i$) at the source and destination nodes.
12: **for** every completed file transfer request $< f_\ell, v_d >$ **do**
13: Update the available network bandwidths along the paths chosen to complete the request

At step 6, the Path Determination method is invoked to select multiple paths for the transfer request. The output from this method makes up the schedule for the

request. The next step (step 7) is to compute the expected minimum completion time for transferring a chunk of the requested file. The transfer completion time is computed as follows. We first divide the file into K portions where K is the number of selected paths. The size of the portion is chosen to be in the same ratio as that of the bottleneck bandwidth of the paths. The transfer completion time is then simply the maximum of the times taken to send each portion along its designated path. At step 9, following the well-known MinMin [42] algorithm, among all the pending requests, the file transfer request with the minimum expected completion time is chosen to be scheduled on the set of resources which yield its minimum completion time. The overall process repeats until all the file transfers have been scheduled.

7.2 GridFTP with Path Splitting and Multi-Pathing

We now discuss the design and implementation of a modified GridFTP server/client infrastructure, which incorporates the multi-hop path splitting and multi-pathing optimizations for file transfer. Our goal is to design a framework which will allow GridFTP clients to benefit from path splitting and multi-pathing in a transparent manner, i.e., clients will issue GridFTP file transfer commands in much the same way as they do now.

The path splitting and multi pathing optimizations require information about the overlay graph of GridFTP servers in order to make efficient routing decisions and to leverage existing "network parallelism" by using multiple data flows on distinct paths. We describe a decentralized system to construct and maintain the overlay graph of GridFTP servers in Section 7.2.2. Our optimized GridFTP client first contacts the source GridFTP server (specified by the user) and retrieves the

overlay graph information. The overlay information is basically a set of tuples $< source, destination, bandwidth >$. Once the GridFTP client has the required information, it runs the path determination algorithm algorithm to determine the multi-hop path(s) through which the file is to be transferred. Implementation of path-splitting in GridFTP involves the modification of the GridFTP server since it involves forwarding of data through intermediate hops. However, multi-pathing can be incorporated by just modifying the GridFTP client globus-url-copy. The GridFTP client, globus-url-copy, currently has support for accepting a set of source-destination url pairs and realizing the file transfers corresponding to them. We employ this functionality to realize multi-pathing. Once a set of paths have been determined by a decision algorithm, globus-url-copy creates a new list of urls along with partial offsets and lengths associated with each url and then performs the file transfer using the multiple designated paths to the destination. In the next section, we present how we have implemented path splitting in GridFTP.

7.2.1 Implementing Path Splitting in GridFTP

GridFTP uses a Data Storage Interface (DSI) to interact with the storage system. The DSI layer accepts requests (e.g., get, put and stat) and performs the required operations on the storage system. To achieve the split-TCP effect, the DSI module must be modified to perform different operations based on the data routing requirements. For instance, a GridFTP server DSI *put* could either transfer the data to the underlying disk subsystem or it could act as a split-point and transfer the data to another GridFTP server. Rizk et al. [66] have implemented a DSI interface to achieve the split-TCP functionality. Their implementation enables a GridFTP client

to specify a multi-hop transfer between a source and a destination URL through a series of intermediate hosts by specifying split URLs. A split URL is essentially a concatenation of multiple normal URLs. For example, a GridFTP client command issued with a source URL A/B and the destination URL C/D means that the file will be transferred from A to D via B and C. Using this DSI for split-TCP functionality requires the user to define the overlay route required.

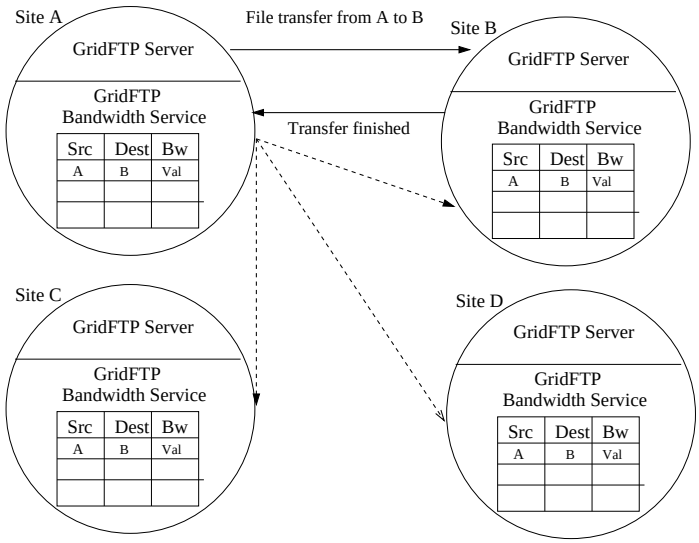


Figure 7.4: A set of GridFTP servers forming an overlay and sharing point-point bandwidth information.

We have made a number of improvements to their work. In their implementation, a GridFTP server could either act as an end point or as an intermediate server but not both, which is very restrictive from the point of view of production-use GridFTP servers. We have extended the DSI layer by allowing it to simultaneously act as an end-point and an intermediate "hop" for different connections. In addition, we have also provided support for directory operations using intermediate servers, a feature not provided in their work [66]. Furthermore, their implementation worked only in the

non-secure mode. We have incorporated the necessary changes to make the split-TCP work with GSI security mechanism.

7.2.2 Constructing Overlay Graph

To obtain the overlay graph information, each GridFTP server acts as a resource provider, thereby exposing WS resource properties (e.g., the bandwidth achieved by file transfers using that GridFTP server). We propose a decentralized service-oriented architecture to facilitate sharing of bandwidth data among sites. Figure 7.4 shows this architecture.

Each GridFTP site also runs a GridFTP bandwidth reporting service. The bandwidth reporting service works in conjunction with the GridFTP server running at the site. For every file transfer which happens through the GridFTP server, the GridFTP server contacts the bandwidth reporting service with the three tuple $< source, destination, bandwidth >$ associated with the transfer. The bandwidth reporting service sends the tuple to each of the peer sites which comprise the overlay. Each of the sites maintains a circular queue of bandwidth values for each source-destination pair. The circular queue stores historical bandwidth information which is used to make predictions about bandwidths in the subsequent time intervals. In this way, each of the sites maintains a set of three tuples $< source, destination, bandwidth >$ corresponding to peer GridFTP sites in the Grid. The bandwidth reporting service also exposes API which allow external entities (e.g. GridFTP clients) to access the bandwidth information and use it to optimize routing and multi-pathing decisions. We employ a Globus toolkit component C-WS core [1], to develop and deploy the bandwidth reporting service at each site.

7.3 Experimental Results

We compare Algorithm 7.4 against our proposed dynamic scheduling algorithm explained in Chapter 6 as well as a baseline strategy that we refer to as *Naive Scheduling*. In this approach, each destination site picks a randomly chosen replica source for retrieving a file instead of employing dynamic bandwidth information or multiple replicas. Here onwards, we refer to our previously proposed scheduling algorithm (Chapter 6) as *GDS* (Global Dynamic Scheduler), the scheduling variant that incorporates path-splitting and multi-pathing optimizations as *GDS-MHMP*, and the scheduling variant that incorporates the modeling of shared bottleneck on top of these two as *GDS-MHMP-SB*.

We use the Globus implementation of GridFTP to transfer files [11]. GridFTP exposes a set of API calls [6] for setting TCP buffer sizes and to obtain portions of a file from a source. For batch transfers, a master scheduler sends control information to clients (destination hosts). The destination hosts then call *globus_ftp_client_partial_get()* to inform a source of the file it needs along with the start and end offsets. This is followed by a series of asynchronous $globus\_ftp\_client\_register\_read()$ calls which are used to transfer data from the source.

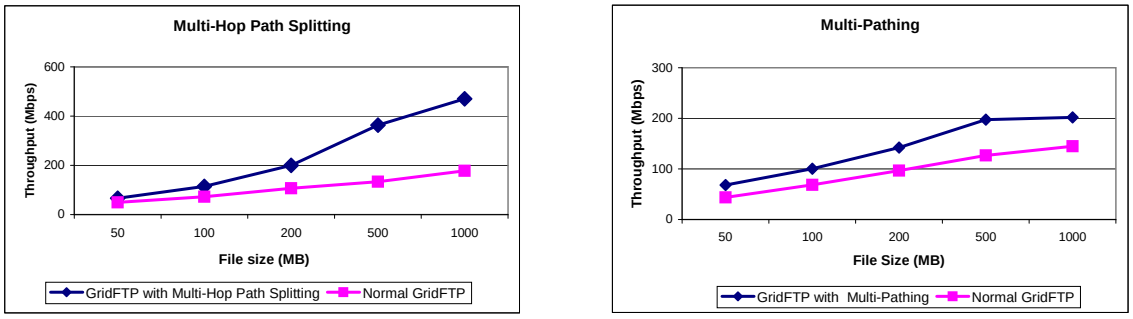
The experiments were carried out across five geographically distributed sites: BMI, a memory/storage cluster in the Department of Biomedical Informatics at the Ohio State University; ST, the Starlight site in Chicago; JA site in Japan which is a part of the Japan Gigibit Network II (JGN2) project; ORNL, which consists of 28 dual processor 3.06 GHz Intel Xeon sites; and ANL, a IA-32 Linux cluster which consists of 96 dual-processor Intel Xeon sites. The latter two sites are on the Teragrid [69]

	BMI	ORNL	ST	JA	ANL
BMI	880	100	200	70	4
ORNL	100	900	800	100	10
ST	80	900	900	800	150
JA	40	120	800	900	8
ANL	4	300	500	30	700

Table 7.1: Link bandwidths (in Mbps) between every pair of sites.

network. Table 7.1 shows the bandwidths in Mbps (Megabits per second) between pair of sites of different sites.

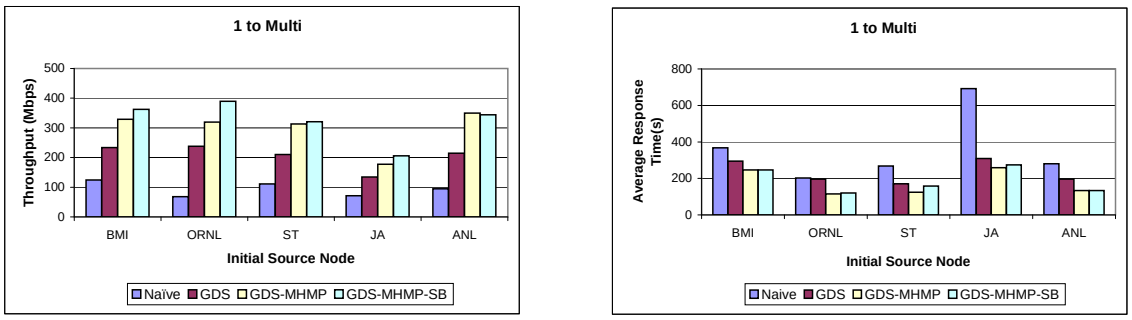
For evaluation, we compared the performance of the various scheduling schemes under a set of well-known communication patterns. For the experimental workloads, we employed files of size 100MB.



(a) (b)

Figure 7.5: (a) Performance improvement due to multi-hop path splitting using the path JA-ST-ANL as compared to using the default path JA-ANL, (b) Performance improvement due to multi-pathing by employing the paths BMI-ORNL-JA and BMI-ST-JA in parallel as compared to using the default path BMI-JA.

Figures 7.5(a) and 7.5(b) highlight the performance improvement due to employing multi-hop path splitting and multi-pathing, respectively. Figure 7.5(a) compares the performance of a file transfer from the JA site to the ANL site using the direct

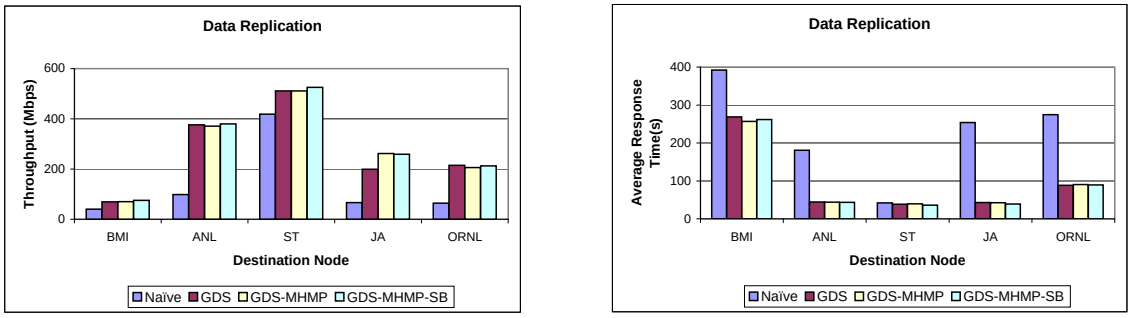


(a) (b)

Figure 7.6: Performance of all the algorithms under the 1-to-all communication pattern in terms of the (a) Average throughput and (b) Average response time.

path as governed by underlying routing, with the case when the ST site is employed as an intermediate path-splitting site. The results show that the multi-hop path performs significantly better than the direct transfer, especially at very large file sizes. This is because, the bandwidth of both JA-ST and ST-ANL is higher as compared to JA-ANL while the end-to-end latency in both the cases is similar. Therefore, the overall throughput improves. Figure 7.5(b) compares the performance of a file transfer from the BMI site to the JA site using the direct path, with the case when the file is split at the BMI site and sent across two independent paths BMI-ORNL-JA and BMI-ST-JA. The results show that with multi-pathing, performance improves by up to 55%. This is because, by employing independent paths, the aggregate bandwidth at the destination site increases.

Figure 7.6 shows the performance of the scheduling schemes under a 1-to-all communication pattern. In this experiment, only one of the 5 sites acts as a source initially and stores all the files. Each of the file needs to be transferred to all the destination sites. This experiment involved transferring 40 100MB files from the source site to each of the destination sites. The results show that *GDS-MHMP* leads to significant



(a) (b)

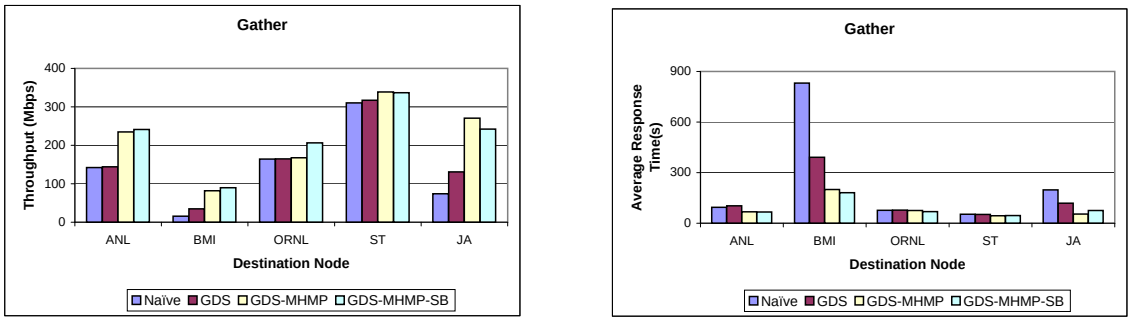
Figure 7.7: Performance of all the algorithms under the effect of data replication, in terms of the (a) Average throughput and (b) Average response time.

improvements in the achieved throughput over the *GDS*. This is because, with only a single source present initially, *GDS* is initially restricted to only using the default path between the source and the destinations. As more replicas of a file get created, those can also act as replica sources thereby giving more flexibility to *GDS*. *GDS-MHMP*, on the other hand, exploits path splitting and multi-pathing and thereby provides, significant performance improvement. An example instance of multi-hop path splitting being, the file transfer from the BMI site to the JA site via the ST site provides 60% improvement over a direct transfer from the BMI site to the JA site for the file size under consideration. Similarly, an example instance of multi-pathing is the scenario which involves the transfer of a file from the BMI site to the JA site. Multi-pathing results in transferring disjoint pieces of file simultaneously across the two paths BMI-ORNL-JA and BMI-ST-JA. The resulting performance improvement over a direct transfer over the default path is around 39%. *GDS-MHMP-SB* performs similar or better in comparison to *GDS-MHMP*. The maximum performance improvement achieved due to modeling the shared bottleneck occurs in the case when ORNL is the initial source site. In this case, the transfer of some of the files to the ST site

and the ANL site (by employing ST site as the source) finish earlier and subsequent transfer of those files to the BMI site employs all the three sources in parallel, that is, the ANL site, the ST site and the ORNL site. However, Paths ORNL-BMI, ST-BMI share common bottlenecks and so do the paths ORNL-BMI and ANL-BMI thereby leading to improved performance for *GDS-MHMP-SB*. The results also show that the proposed scheduling schemes are able to consistently outperform the *Naive Scheduling* scheduling approach. This is because, in the *Naive Scheduling* approach, clients act independently and make requests for files without any coordination. Each file needs to be sent to multiple different destinations, thereby leading to increased end-point contention due to multiple simultaneous requests for the same file. In terms of the average response time, *GDS* and its variants outperform *Naive Scheduling*. This is because, *GDS* schedules the requests with the minimum expected completion time first. On the other hand, in *Naive Scheduling*, since multiple clients act independently of each other, therefore, requests with higher expected completion times can possibly execute before requests with lower expected completion times, thereby, increasing the overall response time.

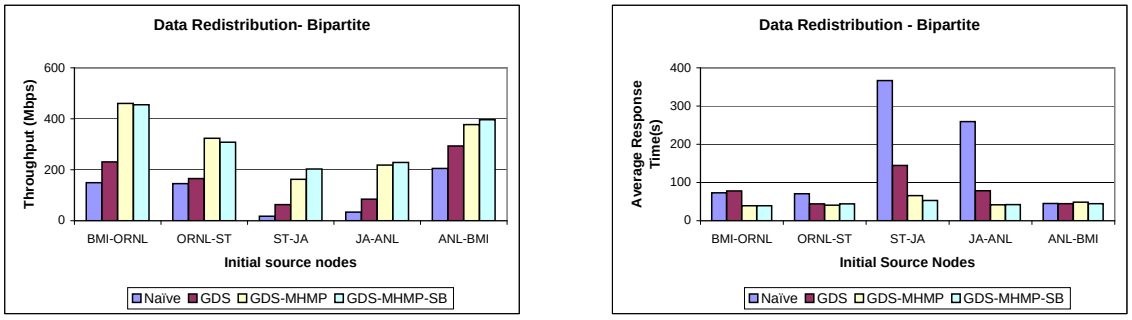
Figure 7.7 shows the performance of the scheduling schemes under the influence of file replication. In this experiment, each of the sites except one, stores a copy of each file. One of the sites act as the destination site to which all the files need to be transferred. Therefore, there are 4 file replicas and the fifth site acts as a destination. This experiment involved transferring 40 100MB files to the destination site. The results show that the scheduling schemes *GDS*, *GDS-MHMP* and *GDS-MHMP-SB* perform fairly similar. This happens because of the existence of multiple initial file replicas, due to which *GDS* makes the choice for the best replica to realize each file

transfer. Therefore, the extent of performance improvement due to path-splitting or multi-pathing is very minimal.



(a) (b)

Figure 7.8: Performance of all the algorithms under a all-to-1 gather file transfer pattern in terms of the (a) Average throughput and (b) Average response time.



(a) (b)

Figure 7.9: Performance of all the algorithms under a bipartite data redistribution pattern in terms of the (a) Average throughput and (b) Average response time.

Figure 7.8 shows the performance of the scheduling schemes for a all-to-1 gather file transfer operation. In this experiment, a set of files are distributed in a round-robin fashion across 4 of the sites, and one of the sites is employed as a destination for all the files. Each file has only replica in the beginning, that is, each file is present on only

one of the sites. The performance results show that in some cases, *GDS-MHMP* and *GDS-MHMP-SB* perform significantly better than *GDS* because these algorithms are able to exploit path splitting and multi-pathing to explore higher bandwidth paths, thereby improving the throughput, while in other cases, the performance improvement is relatively less. This is because of employing a single destination site in each case, which means that the opportunities for path-splitting and multi-pathing are limited.

Figure 7.9 shows the performance of the scheduling schemes for a file pattern which involves data redistribution from a set of source sites to a set of destination sites. In other words, the file transfer pattern is bipartite. Files are distributed in a round-robin order on two of the 5 sites, and the data needs to be remapped to the other three sites in a round-robin manner. The results show that *GDS-MHMP* and *GDS-MHMP-SB* consistently outperform *GDS* in all the cases. This is because, multiple sites collectively act as destinations for a bunch of files, thereby creating more opportunities for path-splitting and multi-pathing in each scenario.

7.4 Conclusion

In this chapter, we explored two key optimizations, namely, multi-hop path splitting and multi-pathing to improve the performance of file transfers over shared wide-area networks. We presented a path determination algorithm which integrates the aforesaid optimizations in order to improve achievable file transfer throughput for a single file transfer. We incorporated this with our previously proposed wide-area scheduling algorithm by making it path-splitting and multi-pathing aware. We also presented the design and implementation of service-oriented architecture which incorporates these ideas within the well known file transfer protocol, GridFTP. Finally,

we looked at certain well-known communication patterns, experimentally analyze the performance of the proposed algorithm on those patterns and show its effectiveness on a wide-area testbed. We observed that the proposed algorithm yields significant performance improvements for communication patterns like 1-to-all broadcast, all-to-1 gather, data redistribution. However, for scenarios which involve data replication, we observed that the improvements are not that significant.

The next chapter outlines our conclusions and proposes some directions for future work.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Scheduling is a well-researched topic. Most of the existing batch schedulers focus on compute-intensive jobs running at supercomputer centers. They take into account CPU related metrics (e.g., user estimated job run times) and system state (e.g., queue wait times) to make scheduling decisions, but they do not take into account data related metrics. However, with the advent of advanced sensing technologies that can rapidly capture data at high resolutions, applications are becoming more and more data-driven. In other words, many current and emerging applications involve accessing and processing huge amounts of data. To address the growing computational and storage requirements of these applications, huge parallel systems, supercomputers, large-scale compute and storage clusters with massive computing power and storage have been built. Effective resource management and scheduling of such systems is critical to satisfying the growing computation/data demands of these applications. Scheduling in this context, requires solving two key problems. One of them is the data staging problem which involves the staging of data from the storage sites to the computational sites where the data analysis needs to be performed. Effectively solving this problem requires coordination of data movement across multiple source

sites, destination sites and intermediate locations, and among multiple users and applications. The other one is the job mapping problem which involves the mapping of data analysis jobs to compute resources in such a manner so as to minimize the completion time of the jobs. Mapping of jobs to compute sites needs to be performed in such a way so as to minimize the data staging cost while at the same time attaining load balance across different computing sites.

8.1 Summary of Research Contributions

In this dissertation, we have developed new scheduling strategies for the aforementioned job scheduling and data staging problem by taking into account the effects of data staging, end-point contention and data locality.

8.1.1 Locality aware job mapping and data replication

For the job mapping problem, our proposed approach formulates the sharing of files (batch-shared I/O) among jobs as a hypergraph. The proposed mapping scheme is based on the partitioning of this hypergraph, thereby mapping a batch jobs to compute nodes in such a way that the overheads of data transfer are minimized while load balance across nodes is maintained. The proposed approach takes into account global job-file sharing information while making mapping decisions, thereby outperforming local greedy heuristics like MinMin, MaxMin and Sufferage. We extended this approach to an online scenario where jobs dynamically arrive over time. Specifically, we construct a hypergraph corresponding to the snapshot of the system at each scheduling event. The hypergraph models 1) the current state of the system that includes the pending jobs and the files requested by them, 2) the currently executing jobs, and 3) the files already cached on the compute nodes due to previously executed jobs.

This is followed by K-way partitioning of the hypergraph to obtain a load-balanced cut minimizing mapping of the pending jobs onto the compute nodes. The currently executing jobs are incorporated in the partitioner to take into account the current value of load on each of the compute nodes and thereby facilitate load balance as a result of the new partitioning. We show that our proposed online scheduling approach achieves performs better than previously proposed heuristics - *MET*, *MCT*, *SA* and *JobDataPresent* with *Data Least Loaded* - when there is a high degree of file sharing among jobs. Under high load situations, the proposed approach, which takes an integrated view of scheduling of computation and data placement, outperforms the other heuristics significantly.

Furthermore, we propose two strategies for simultaneous scheduling and replication - one approach formulating the problem of coordinating scheduling and replication using a 0-1 Integer Programming and another using a bi-level hypergraph partitioning strategy. Both approaches also model disk storage space constraints at the compute cluster. Our evaluations show that our strategies achieve significant performance improvement over *MinMin* with *Implicit replication* and *JobDataPresent* with *Data Least Loaded*. The base schemes do not explicitly consider inter-job dependencies arising out of file-sharing and thus make local decisions based on greedy heuristics. Among the proposed algorithms, the IP formulation results in the best batch execution time. However, it suffers from high scheduling time. The partitioning approach results in slightly longer batch execution times, but is much faster than the IP based approach. We conclude that the IP based approach is attractive for small workloads, while the partitioning approach is preferable for large scale workloads and system configurations.

8.1.2 Data transfer scheduling for data-centers and wide-area environments

For the data staging problem, we address it both in the context of data-centers as well as distributed wide-area environments. For data-centers, we proposed two strategies for collectively scheduling a set of file transfer requests made by a batch of data-intensive jobs on heterogeneous systems - one approach formulates the problem using 0-1 Integer Programming and another based on using max-weighted graph matching. The performance results show that the IP formulation results in the best overall file transfer time. However, it suffers from high scheduling time. The graph matching based approach results in slightly higher file transfer completion times, but is much faster than the IP based approach. Moreover, the matching based approach is able to match the performance of the Insertion scheduling approach with a much lower scheduling overhead. We conclude that the IP based approach is attractive for small workloads, while the matching based approach is preferable for large scale workloads.

For the wide-area context, we proposed a dynamic scheduling algorithm which schedules a set of data transfer requests made by a batch of data-intensive jobs. We also proposed a network flow based integer programming formulation of the scheduling problem, which is used to find a lower bound on the transfer time under idealistic conditions of resource availability and performance. The proposed scheduling algorithm is adaptive in that it accounts for network bandwidth fluctuations in the wide-area environment. The algorithm incorporates simultaneous transfer of disjoint chunks of the same file from different replica sources to a destination node, thereby increasing

the aggregate bandwidth. Adaptive replica selection is used for transferring different chunks of the same file by taking dynamic network information into account. We employ GridFTP for data transfers and utilize information from past GridFTP transfers to perform predictive bandwidth estimations. We have shown the effectiveness of our algorithm through experimental evaluations on a wide-area testbed. In general, the proposed algorithm is expected to provide greater benefits with increasing degree of data replication. With a very low degree of replication, the algorithm is restricted in its choice of multiple replicas, thereby not giving significant performance improvements. Moreover, it is expected to perform well for multi-destination workloads. This is because, it, can dynamically take into account the existence of new replicas as some of the files are transferred to their respective destination nodes, and employ those replicas for subsequent file transfers. Furthermore, we extend our proposed dynamic scheduling algorithm by incorporating the effects of multi-hop path splitting and multi-pathing to improve wide-area file transfer performance. We proposed a path determination algorithm which integrates the aforesaid optimizations in order to improve achievable file transfer throughput for a single file transfer. We observed that the proposed algorithm yields significant performance improvements for communication patterns like 1-to-all broadcast, all-to-1 gather, data redistribution.

8.2 Future Research Directions

In this dissertation, we have proposed data-locality aware scheduling schemes which can improve the performance of data-intensive applications on distributed platforms. However, there are several interesting research topics that still need to be explored.

8.2.1 QoS-aware scheduling

A scientific collaboration like high-energy physics is representative of a distributed data-intensive computing paradigm where a set of compute, storage and network resources are used in a collective fashion to advance science. The number of scientists involved in such collaboration can be 100-1000s. A typical job request by a scientist involves staging of data from storage sites to the computation sites followed by computation on the staged data. However, there is no way for a scientist to specify the urgency of his job to the job scheduler. For example, the scientist may wish to have the data-staged by a certain time. This becomes important especially with the increasing size of datasets which is already in petabytes. Subsequently, the scientist may want the results of his experiment by a specified time. Projects like the LambdaStation [4] will enable the scientists the usage of advanced optical networking technologies through on demand network-provisioning. Therefore, it will enable the scientists to request for differentiated services based on their QoS requirements.

The job mapping and the data-staging algorithms proposed in this dissertation can be extended to incorporate QoS constraints imposed by the user submitting the job. The QoS constraints can be specified in terms of the deadlines on the job completion time or the data-staging time etc.

There are several issues which need to be addressed in order to solve this problem.

1. **Optimization metric**

   First is the choice of the metric to be used for optimization. An obvious metric for optimizing the batch execution for jobs submitted by a single user is the overall batch turnaround time. However, for a batch of jobs submitted by

multiple users, a better metric is to choose the average response time since the batch execution time is not important in this context. The batch of jobs under consideration also has a varying QoS requirements. Some jobs have hard real-time deadlines, some have soft QoS requirements which means the jobs are executed to completion even if the specified deadlines are not met , while others do not have QoS any constraint at all. Another metric which therefore, needs to be considered is the number of hard real-time jobs which miss their deadlines. A third metric, is the average excess time taken by the jobs with soft QoS requirements over their specified deadlines in case they do not finish by the deadline. The scheduling scheme therefore should prioritize these three metrics and try to achieve the best value for the primary metric while also trying to perform good for the other metrics.

2. **Job input information**

   Typically, supercomputer centers for compute jobs allow users to specify a description of their job in terms of the expected execution time, number of processors for a parallel job etc. In a data-intensive application scenario, a job can be expected to provide the scheduler with the following information.

   - Expected Computation time
   - Set of files to read
   - Set of files to write
   - May or may not submit information like computation/data ratio
   - deadlines (hard or soft)

Since we are focusing on batches with QoS requirements, it becomes all the more imperative to measure the expected completion time of a job as accurately as possible. This in turn, requires an accurate modeling of the contention at the storage nodes and in the network which are used to transfer the job data. Another interesting research issue is to measure the effectiveness of the scheduler with varying granularities of information provided as job input. For example, a job specifying its computation/data ratio stands a better chance of meeting its requirements since it aids the scheduler in making better and more accurate decisions.

8.2.2 Incorporating data locality into existing compute intensive schedulers

Compute-intensive job schedulers provide very simple mechanisms to incorporate the data staging requirements of jobs. Data locality-aware scheduling algorithms can potentially reduce wasted compute cycles and make use of file affinity when executing jobs. SLURM is an open source compute intensive batch scheduler that provides flexibility in terms of incorporating new scheduling policies. The file-locality concepts proposed in this dissertation can be incorporated into SLURM to make it data-aware. This will entail addressing key issues like incorporating a data specification language into SLURM, enhancing a compute node's state information to include locality, enhancing the API to incorporate the new set of attributes that will be added to the node state and the job state etc. The benefits of this research would span across many scientific communities which collect data from scientific instruments and move the data to compute clusters for analysis.

8.2.3 Distributed scheduling for data-intensive jobs

Most of our prior work has employed a centralized batch scheduler to make job scheduling and dispatching decisions. A key disadvantage of using a centralized scheduling system is that it becomes a critical single point of failure, more so in environments where resource availability is uncertain. In addition, a centralized design may not be the best one from a scalability perspective. Therefore, one of the key modifications in the design of the scheduler is to have a distributed scheduling system where there are multiple distributed schedulers, each managing their local sites. Each site-level scheduler will perform scheduling decisions for jobs submitted to that site. A resource allocation request comprising of more than sites will require communication among the site-level schedulers in order to make a coordinated job allocation decision. A key tradeoff which needs to addressed in this regard, is the scalability benefit due to using a distributed scheduling mechanism and the lack of global information which is available in the centralized case.

8.2.4 Optimizing data transfers for P2P file-sharing systems

Peer-to-peer file transfer systems like BitTorrent facilitate exchange of files among millions of users on the Internet. Each individual user or peer stores some files on its local machine and exchanges those files with other peers through HTTP-style protocols. Each peer can act as a sender for some files while acting as a receiver for other files. These systems typically employ incentive-based file sharing where in the peers which contribute more data at faster rates get preferential treatment for downloads. The wide-area data transfer scheduling algorithms proposed in this work also aim at optimizing data transfers but with a different optimization perspective, that is,

the total transfer time of request. A potential research direction is to explore the applicability of the proposed data-transfer scheduling concepts in a P2P file transfer setting.

8.2.5 Communication minimization parallel job mapping

Previous approaches to topology-aware mapping of jobs to processors have primarily focused on minimizing a weighted distance metric, which aims at mapping heavily communicating jobs on nearby processors. However, with techniques like wormhole routing, the no-load latency among distantly located processors is not significantly different from closely placed processors. In such cases, congestion is a better determinant of application execution time. An interesting research direction is to look at congestion-based metrics to evaluate mapping schemes. The key idea is to develop mapping schemes which look at congestion as the objective function and to evaluate the schemes on parallel architectures like BlueGene/L.

BIBLIOGRAPHY

[1] GT C WS core. http://www.globus.org/toolkit/docs/4.0/common/cwscore/.

[2] The BaBar experiment. http://www.slac.stanford.edu/BFROOT/.

[3] The DZero experiment. http://www-d0.fnal.gov.

[4] The LambdaStation project. http://www.lambdastation.org/.

[5] The Large Haldron Collider (LHC) . http://lhc.web.cern.ch/lhc/.

[6] Globus ftp client api. *http://www.globus.org/api/c/globus_ftp_client/html/index.html*, 2002.

[7] Ilog cplex 9.1. 2004. http://www.ilog.com/.

[8] Globus metrics, version 1.4. *http://incubator.globus.org/metrics/reports/2007-02.pdf*, 2007.

[9] W. Allcock, I. Foste, and R. Madduri. Reliable data transport: a critical service for the grid. In *Building service based grids workshop, Global Grid Forum*, 2004.

[10] William Allcock. Gridftp: Protocol extensions to ftp for the grid. In *Global Grid ForumGFD-R-P.020*, 2003.

[11] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The globus striped gridftp framework and server. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. Fort Lauderdale, FL, April 2002. Also available as University of Maryland Technical Report CS-TR-4290 and UMIACS-TR-2001-68.

[13] A. Bakre and B. R. Badrinath. I-tcp: indirect tcp for mobile hosts. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems*, page 136, Washington, DC, USA, 1995. IEEE Computer Society.

[14] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1998.

[15] Alessandro Bassi, Micah Beck, Terry Moore, James S. Plank, Martin Swany, Rich Wolski, and Graham Fagg. The internet backplane protocol: a study in resource sharing. *Future Gener. Comput. Syst.*, 19(4):551–562, 2003.

[16] M. Beck, T. Moore, J. S. Plank, and M. Swany. Logistical networking: Sharing more than the wires. In S. Hariri, C. A. Lee, and C. S. Raghavendra, editors, *Active Middleware Services*, Norwell, MA, 2000. Kluwer Academic.

[17] John Bent, Doron Rotem, Alexandru Romosan, and Arie Shoshani. Coordination of Data Movement with Computation Scheduling on a Cluster. In *Workshop on Challenges of Large Applications in Distributed Environments (CLADE2005)*, pages 25–34, Research Triangle Park, NC, July 2005. IEEE Computer Society Press.

[18] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve open source mixed integer linear programming system. In *Version 5.5.0.7*, May 2005. http://lpsolve.sourceforge.net/5.5/.

[19] D. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, J. Fox, P. andGarcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams. The earth system grid: supporting the next generation of climate modeling research. In *Proceedings of the IEEE*, volume 93, pages 485–495, 2005.

[20] Biomedical Informatics Research Network (BIRN). http://www.nbirn.net.

[21] Shahid Bokhari, Benjamin Rutt, Pete Wyckoff, and Paul Buerger. An evaluation of the OSC FAStT600 Turbo storage pool. Technical Report OSUBMI_TR_2004_n02, Dept. of Biomedical Informatics, The Ohio State University, 2004. http://web.bmi.ohio-state.edu/resources/techreports/OSUBMI_TR_2004_n02.pdf.

[22] D. L. Brown, Geoffrey S. Chesshire, William D. Henshaw, and Daniel J. Quinlan. Overture: An object oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

[23] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proc. of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pages 430–441, 2001.

[24] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, Cancun, Mexico, 2000. IEEE Computer Society.

[25] Umit Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Systems.*, 10(7):673–693, 1999.

[26] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/∼umit/software.htm, 1999.

[27] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel H. Saltz. Titan: A high-performance remote sensing database. In *Proc. of the 13th International Conference on Data Engineering (ICDE 1997)*, pages 375–384, Washington, DC, USA, 1997. IEEE Computer Society.

[28] A Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe. Wide area data replication for scientific collaborations. In *Proceedings of Grid 2005 - 6th IEEE/ACM International Workshop on Grid Computing*, Seattle WA, November 2005.

[29] Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: a framework for constructing scalable replica location services. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[30] Ann L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman, and Robert Schwartzkopf. Performance and scalability of a replica location service. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 182–191, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The neos server. *IEEE Comput. Sci. Eng.*, 5(3):68–75, 1998.

[32] Frederic Desprez and Antoine Vernois. Simultaneous Scheduling of Replication and Computation for Bioinformatic Applications on the Grid. In *Workshop on Challenges of Large Applications in Distributed Environments (CLADE2005)*, pages 66–74, Research Triangle Park, NC, July 2005. IEEE Computer Society Press.

[33] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[34] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS 1989)*, pages 247–252, New York, USA, 1989.

[35] Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Math. Program.*, 104(1):91–104, 2005.

[36] L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6(3):419–433, 1958.

[37] Harold N. Gabow. An efficient implementation of edmonds' algorithm for maximum matching on graphs. *J. ACM*, 23(2):221–234, 1976.

[38] Arnaud Giersch, Yves Robert, and Frédéric Vivien. Scheduling tasks sharing files from distributed repositories. In *Euro-Par 2004: Parallel Processing: 10th International Euro-Par Conference, volume 3149 of LNCS*, pages 246–253, September 2004.

[39] Grid Physics Network (GriPhyN). http://www.griphyn.org.

[40] Robert L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[41] Ian Holyer. The np-completeness of edge-colouring. *SIAM Journal on Computing*, 10(4):718–720, 1981.

[42] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. volume 24, pages 280–289, New York, NY, USA, 1977. ACM Press.

[43] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Heuristics for scheduling i/o operations. volume 8, pages 310–320, Piscataway, NJ, USA, 1997. IEEE Press.

[44] Avi Kavas, David Er-El, and Dror G. Feitelson. Using multicast to pre-load jobs on the parpar cluster. *Parallel Computing*, 27(3):315–327, 2001.

[45] Kamer Kaya. Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments. *IEEE Trans. Parallel Distrib. Syst.*, 17(8):883–896, 2006. Member-Cevdet Aykanat.

[46] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, P. Sadayappan, and Joel Saltz. Scheduling file transfers for data-intensive jobs on heterogeneous clusters. In *Proceedings of Europar 07, The 13th European Conference on Parallel and Distributed Computing*, Rennes, France, 2007. To appear.

[47] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, P. Sadayappan, and Joel Saltz. Scheduling file transfers for data-intensive jobs on heterogeneous clusters. Technical Report OSU-CISRC-1/07-TR05, CSE Dept., The Ohio State University, 2007.

[48] Gaurav Khanna, Tahsin Kurc, Umit Catalyurek, Rajkumar Kettimuthu, P. Sadayappan, and Joel Saltz. A dynamic scheduling approach for coordinated wide-area data transfers using gridftp. In *Proc. of 22th International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, Florida, 2008. to appear.

[49] Gaurav Khanna, Tahsin Kurc, Umit Catalyurek, P. Sadayappan, and Joel Saltz. A data locality aware online scheduling approach for i/o-intensive jobs with file sharing. In *Proceedings of the 12th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2006)*, 2006.

[50] Gaurav Khanna, Nagavijayalakshmi Vydyanathan, Umit Catalyurek, Tahsin Kurc, Sriram Krishnamoorthy, Joel Saltz, and P. Sadayappan. Task scheduling and file replication for data-intensive jobs with batch-shared i/o. In *Proceedings of the The 15th IEEE International Symposium on High Performance Distributed Computing (HPDC'06)*, 2006.

[51] Gaurav Khanna, Nagavijayalakshmi Vydyanathan, Tahsin Kurc, Umit Catalyurek, Pete Wyckoff, Joel Saltz, and P. Sadayappan. A hypergraph partitioning based approach for scheduling of tasks with batch-shared i/o. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 792–799, Washington, DC, USA, 2005. IEEE Computer Society.

[52] G. Kola and M. Livny. Diskrouter: A flexible infrastructure for high performance large scale data transfers. Technical Report CS-TR-2003-1484, University of Wisconsin, 2003.

[53] Tevfik Kosar and Miron Livny. Stork: Making data placement a first class citizen in the grid. In *ICDCS '04: Proc.of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 342–349, Washington, DC, USA, 2004. IEEE Computer Society.

[54] David Kotz. Disk-directed i/o for mimd multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, 1997.

[55] Sriram Krishnamoorthy, Umit Catalyurek, Jarek Nieplocha, and P. Sadayappan. An approach to locality-conscious load balancing and transparent memory hierarchy management with a global-address-space parallel programming model. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, Rhodes Island, Greece, 2006. to appear.

[56] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling distributed applications: the simgrid simulation framework. In *Proc. of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 138–145, 2003.

[57] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.

[58] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra A. Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, San Juan, Puerto Rico, 1999. IEEE Computer Society.

[59] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 104–113, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[60] Manish Mehta, Valery Soleviev, and David J. DeWitt. Batch scheduling in parallel database systems. In *Proceedings of the International Conference on Data Engineering*, pages 400–410, Vienna, Austria, April 1993.

[61] H.H. Mohamed and D.H.J. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In *2004 IEEE International Conference on Cluster Computing*, pages 287–298. IEEE Society Press, 2004.

[62] S. Parthasarathy and R. Subramonian. Facilitating data mining on a network of workstations. In *Proc. of the IASTED International Conference on Parallel and Distributed Systems*, Boston, 1999.

[63] Thomas Phan, Kavitha Ranganathan, and Radu Sion. Evolving toward the perfect schedule: Co-scheduling job assignments and data replication in wide-area systems using a genetic algorithm. In Dror G. Feitelson, Eitan Frachtenberg,

Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 3834 of *Lecture Notes in Computer Science*, pages 173–193. Springer, 2005.

[64] H. Pucha and Y. C. Hu. Overlay tcp: ending end-to-end transport for higher throughput. In *Poster in ACM SIGCOMM*, Philadelphia, PA, 2005.

[65] Kavitha Ranganathan and Ian T. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 2002), Edinburgh, UK*, pages 352–358, 2002.

[66] P. Rizk, C. Kiddle, and R. Simmonds. A gridftp overlay network service. In *In Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, Baercelona, Spain, 2007.

[67] Martin Swany. Improving throughput for grid applications with network logistics. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 23, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Atsuko Takefusa, Osamu Tatebe, Satoshi Matsuoka, and Youhei Morita. Performance analysis of scheduling and replication algorithms on grid datafarm architecture for high-energy physics applications. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.

[69] TeraGrid. http://www.teragrid.org.

[70] Brian Tierney, Jason Lee, Ling Tony Chen, Hanan Herzog, Gary Hoo, Guojun Jin, and William E. Johnston. Distributed parallel data storage systems: a scalable approach to high speed image servers. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 399–405, New York, NY, USA, 1994. ACM Press.

[71] Brian L. Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake Jr. A network-aware distributed storage cache for data intensive environments. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 33, Washington, DC, USA, 1999. IEEE Computer Society.

[72] Mustafa Uysal, Tahsin M. Kurc, Alan Sussman, and Joel Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *Proc. of the 4th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, LNCS, Vol. 1511*, pages 243–258. Springer-Verlag, May 1998.

[73] Sudharshan Vazhkudai and Jennifer M. Schopf. Predicting sporadic grid data transfers. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 188, Washington, DC, USA, 2002. IEEE Computer Society.

[74] Sudharshan Vazhkudai, Jennifer M. Schopf, and Ian T. Foster. Predicting the performance of wide area data transfers. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 270, Washington, DC, USA, 2002. IEEE Computer Society.

[75] Lingyun Yang, J. M. Schopf, and I. Foster. Improving parallel data transfer times using predicted variances in shared networks. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 734–742, Washington, DC, USA, 2005. IEEE Computer Society.

[76] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2003.

[77] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Softw. Pract. Exper.*, 23(12):1305–1336, 1993.