I/O Considerations in Efficient Heterogeneous Data Distributions

by

Mario Nakazawa

(Under the direction of David K. Lowenthal)

Abstract

Large scale scientific applications can take a significant amount of time when run sequentially. Parallelism is one way to reduce their execution time. Two key problems in efficiently running a program in parallel on a distributed memory architecture are (1) scheduling the application and (2) distributing the work so as to simultaneously minimize the time spent in communication and in local processing on each node. The input to these programs can be very large, and they are increasingly likely to run on a heterogeneous architecture—both situations make it likely that they will need to access the disk while running, incurring I/O costs. Ignoring these costs can greatly degrade performance because they are typically an order of magnitude more expensive than message passing latencies or cache misses.

We determined using a simulator that I/O-awareness in gang scheduling can increase throughput and reduce turnaround times of applications by as much as a factor of three. This result motivated us to develop a runtime system that solves the data distribution problem for applications running on a heterogeneous cluster of machines. This dissertation describes two components of this system we implemented: (1) the search mechanism, and (2) the computational model it uses to evaluate each candidate distribution. An exhaustive search is computationally intractable, so we simplified the problem by assuming a monotonic relationship between a data distribution and the resulting application execution time. Our search algorithm, which we call *GBS*, is optimal in solving this simplified problem. We also developed a computation model called MHETA, which is used as an evaluation function by the search algorithm. The model integrates an application's structural information and instrumented measurements to generate a predicted execution time given an input distribution. In our experimental test bed consisting of four scientific applications on 17 emulated architectures, *GBS* on average produces distributions within 5% of the optimal within one second of running. We also show that MHETA is on average at least 97% accurate in its predictions, indicating that the *GBS* algorithm (with MHETA) successfully finds effective data distributions for a parallel application running on a heterogeneous cluster on the fly.

Index words: Computation models, Data distributions, Scheduling, I/O Awareness, Heterogeneous architectures

I/O Considerations in Efficient Heterogeneous Data Distributions

by

Mario Nakazawa

B.A. University of Pennsylvania, 1994

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2005

I/O CONSIDERATIONS IN EFFICIENT HETEROGENEOUS DATA DISTRIBUTIONS

by

MARIO NAKAZAWA

Acknowledgments

I would like to thank my wife Jeanne for her love and support. She knew what she was getting into when we got married in the middle of the dissertation writing process, and her faith in my abilities never wavered. We make a great team.

I also appreciate the efforts of my committee members in helping me correct and write this dissertation. I feel more confident in my future endeavors by their efforts to extend my knowledge of the material and push me to the limits of what I know.

Finally, I wish to thank my adviser Dr. Lowenthal for asking me to work with him those many years ago when I was looking for a topic to investigate. He was always ready and willing to help me in my writing and critical thinking, and he was an invaluable guide to bring my ideas into fruition.

Table of Contents

List of Figures

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Computer programs are a popular means to solve problems that involve complicated calculations. Once created and checked for errors, these programs are generally much faster and more accurate than attempting to do the computation manually. This is particularly true in the realm of scientific computing. Scientists describe the inner workings of natural phenomena with complex theories, which they can use to predict the results of experiments. Such theories consist of highly structured and formally written algorithms and are thus relatively easy to encode into computer programs. Without any regard to its efficiency, unfortunately, a naively designed scientific program can run for a very long period of time. One popular technique to significantly reduce a program's completion time is *parallelization*, in which the computational work is divided and spread between many processing units. By having all units work at the same time, they can process parts of the problem simultaneously, and hence the application can finish faster. We calculate the speedup achieved through parallelization as the ratio of the time taken to run the program sequentially over the time taken when run in parallel. Ideally, we would like a perfect linear speedup, which means the execution time decreases proportionally with the number of processors working on the program. (i.e. with P units, the speedup is P, and the application finishes in $1/P$ of the sequential time).

This ideal is difficult to achieve in general, as there are a variety of problems one must solve to create an efficient parallel application. Some problems are dependent on the structure of the target architecture. At one end of the range is a single multi-processor machine with a single memory that is shared between the processors. Lack of proper synchronization mechanisms in the program can result in race conditions when two processors try to simultaneously manipulate the same variable. Depending on the order in which the data is accessed, the result of the computation can differ, and the application can produce incorrect output. In addition, upgrading components of this large system is often prohibitively expensive, and some features, such as the number of processors, are difficult to change. At the other end of the spectrum is a *distributed* system, in which each machine in a cluster (individually also referred to as a “node”) is a uniprocessor machine with its own memory. Synchronization is typically implemented via message passing, so when two nodes need to access the same variable, they must use explicit communication. Unlike in the multi-processor machine, the amount of aggregate memory scales up as the number of nodes increase. One also can replace, upgrade or add a single machine or component at a time, which is more cost effective than having to upgrade the entire system. These structural characteristics impact how a programmer transforms a sequential program into an equivalent and efficient parallel version. Some of the issues involved to successfully create this parallel version are detailed in Section 1.1.

We assume that the programmer has adequately addressed the problems defined above and has generated the most efficient parallel version of the scientific program. Once the program is ready to execute, two important decisions are (1) the scheduling policy, and (2) the best assignment of the work onto the architecture running the program. Previous work in these last two problems have assumed that the costs to access backup store (the *I/O costs*) are negligible. However, these costs are increasingly difficult to ignore, and this thesis addresses how the I/O costs impact decisions for both when scheduling and distributing work. We started our investigation on the impact of I/O costs on scheduling on a homogeneous cluster. This motivated our change in focus to examine the inclusion of I/O costs when determining an efficient division of work, which will be explored in more detail here. We describe some of the fundamental factors one must consider and the different possible strategies for distributing work in Section 1.2. Section 1.3 presents a general description of the interaction between these issues in more detail, followed by the outline of the dissertation in Section 1.4.

1.1 Issues in Parallelizing Sequential Programs

The simplest implementation of a program is a consecutive sequence of instructions (in what is referred to the *source code*) to run on a single processor. When parallelizing the program, the programmer rewrites these instructions into an equivalent parallel version in two steps. The user first divides the program into phases and identifies those that can execute in parallel. The programmer then must insert functions to coordinate the execution of the phases to ensure that the output of the application is the same as it would be if it were executed sequentially (i.e. is it *correct*). Some of the issues involved at each of these steps are presented in more detail below.

Some phases, such as those that loop over elements of a multi-dimensional array, are conducive to running in parallel—each processing unit is assigned a subset of the iterations. Other phases can consist of loops with high dependencies between iterations—these cannot efficiently run in parallel and so run sequentially. However, an efficient parallel version may exist for an algorithm that initially seems difficult to parallelize, in which case the programmer could replace the original code with the parallel implementation of the algorithm. Another important issue to address is granularity, a measure of the number and size of the phases. An application that consists of many small phases is generally too fine-grained, which is problematic because the overhead of coordinating them starts to dominate execution time. Therefore, the goal is to make sure the phases will have enough work so that the processing units spend an appreciable amount of time executing them before any coordination occurs.

To ensure correctness, communication or synchronization functions are then inserted at the beginning or end of each phase. The exact mechanism used depends on the target architecture. In a shared memory multi-processor, each processor has access to a shared region of memory. Synchronization is usually implemented using locks or semaphores for access to shared variables, but the programmer still must insert barriers or other control mechanisms to properly prevent race conditions. Our work assumes an alternative architecture, a distributed memory system, where each node has its own memory. Synchronization in this case is usually done via explicit message passing, and because non-local data must be transferred this mechanism also incurs communication costs. The Message Passing Interface (MPI) library is commonly used to implement communication primitives (sends or receives), and well-defined communication patterns (for example, a reduction or nearest neighbor communication). The programmer can perform this step by hand or rely on parallelizing preprocessors to automatically insert the MPI function calls where necessary. Regardless of how these functions were inserted, we assume applications have been parallelized to run on distributed memory systems using the MPI mechanism for message passing.

1.2 Dividing the Work

In our work, the nature of a distributed memory architecture on which these applications run is assumed to be unknown until just prior to runtime. As the parallel application executes, the runtime system must be able to partition the work so that it runs efficiently. There are two main approaches to solving this problem. The "task parallel" approach assumes there are several distinct phases (called tasks) that can run in parallel, and different ones are assigned to each node. Each node thereby executes a different set of instructions on the data (possibly the same data) and typically communicates after it finishes each task. The important concern in this case is how to order the tasks on the nodes to satisfy their dependency constraints. We address an alternative technique, called "data parallelism", that decomposes the data domain. In these SPMD (Single Program Multiple Data) applications, all the nodes perform the same set of instructions (phase) on the data assigned to them. All nodes run the same phase simultaneously, so there are no ordering constraints to consider as there are in task parallelism. However, all nodes typically participate in a barrier or some other global synchronization function when they reach the end of the phase to share information they need to proceed to the next phase.

Assuming that each node has no memory capacity problems, there are two complications involved with distributing data of SPMD programs. Of lesser importance is that as the data is divided between increasing number of nodes, the time taken to perform computation per node decreases, but the communication overhead increases. The benefits of further parallelizing the application is offset by the higher communication costs at a certain point. An even more important problem is the large synchronization cost resulting from a poor allocation of data. Taken to an extreme, suppose a single machine is given all the work. The other nodes have essentially no computation, so they reach the synchronization point very quickly. However, the execution time of the program is determined by the completion of the *slowest* node. The completion time of the program itself is slower in this case than if it ran on a single machine because of additional various overheads inherent in parallel execution. A division of data must therefore aim to minimize the maximum of communication and local computation costs of all the nodes simultaneously.

1.3 I/O Costs and Data Assignment

Much of previous research on partitioning and allocating data of an SPMD application considers only computation and communication costs. This simplification is possible due to two typical assumptions:

1. These applications run on homogeneous clusters in which all nodes have similar characteristics, such as processing power.
2. The nodes of the cluster have practically infinite memory capacities.

Indeed, these assumptions are valid for scientific programs running on expensive large scale architectures such as supercomputers with specialized hardware and large memory. However, *heterogeneous* clusters of machines are an increasingly popular alternative due to their cost-effectiveness. They do not rely on expensive hardware, and nodes can easily be added without having to match the configuration of those already on the cluster. A consequence of this shift is that both assumptions are no longer valid. First, the memory sizes of these machines can vary widely, as older machines with smaller memory capacities are networked together with more modern machines with larger capacities. There is now a higher likelihood of a data distribution assigning more data to a node than it can store in memory. Secondly, memory is relatively expensive and small as opposed to secondary storage that is often several orders of magnitude larger and cheaper per byte. Therefore, large-scale scientific programs, such as those that model weather or analyze satellite images, typically work with very large datasets that can often exceed the total aggregate memory capacity of a cluster. Even adding more machines will not overcome this problem; the accuracy of the application output is constrained by the input data size, and it is always possible to raise the resolution to get better results.

Given a data distribution in this computing environment, a node that cannot store the local portion of the working data in memory must save it on disk (sometimes referred as the backup store). The application is then *out-of-core*, and the local dataset must be read from and written to disk in pieces. Hardware optimizations such as write-back buffers allow a call to the write function to return quickly, reducing the effective latency of that operation. Reading data from the backup store is in contrast potentially very expensive because the application must wait for the data to arrive from disk before it can proceed. Ignoring either cost usually results in poor performance, as the speed of memory and processors continue to out-pace the progress of I/O latency reduction. This large disparity is due to the fact that reading data from a magnetic disk involves mechanical operations (movement of the reading arm and the platters in which the data is stored) which are orders of magnitude slower than accessing memory. For this dissertation, a local, single disk drive is assumed, but the backup store can be on a disk of a remote machine; the latency has the potential to be even larger if one adds costs to transfer the data over a network. Our target scientific applications repeat computation over several iterations, and the entire local dataset must be read from and potentially written to disk at each iteration, multiplying the I/O costs.

The consideration of I/O in data distributions introduces an additional trade-off. Data distributions that minimize accesses to disk and avoid expensive latency costs are desirable. As the same time, the computational power of the nodes in a heterogeneous cluster are non-uniform, requiring a data distribution that also minimizes load imbalances. Any benefit from assigning more data to a node with a faster CPU and small memory in order to balance the workload can be offset by the penalty of accessing the disk, resulting in overall poor performance. This penalty is exaggerated if the access speed of the disk is also very slow. We therefore extend the problem of distributing

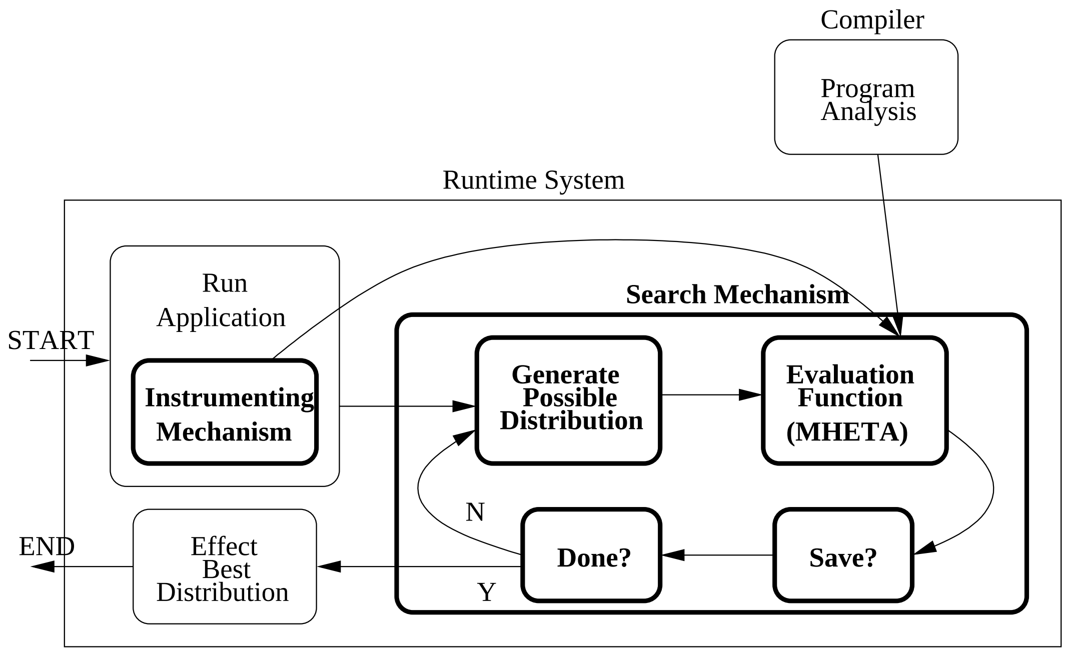


Figure 1.1: The general structure of the SHARED runtime system. Two components, the search mechanism and the MHETA evaluation function, have already been implemented and are detailed in Chapters 4 and 5, respectively. The instrumenting component is partially done by hand, and details for automating both this process and program analysis by the pre-processor are described in Chapter 5.

data to minimize the maximum execution time of an out-of-core application running on a heterogeneous cluster. This time includes the costs to perform computation, communication, *and* disk accesses of all the nodes; this is the *out-of-core heterogeneous data distribution problem*.

Our eventual goal is to develop a runtime system we call SHARED (**S**ystem for **H**eterogeneous **AR**chitecture **E**fficient **D**ata distributions) shown in Figure 1.1 that automatically determines the best distribution of data for large-scale scientific applications. Two main portions of this system, the data distribution search mechanism and its evaluation function, have already been implemented. We first note that an exhaustive search through all possible data distribution candidates is computationally intractable. Therefore, we developed a guided search heuristic we call *GBS* (Generalized Binary Search) that starts from a random candidate which it then continually modifies until no more improvement is possible. The *GBS* algorithm is based on the assumption that there is a monotonic relationship between how much data and work are assigned to a node with the length of time it will take to finish the work. We prove that this algorithm is optimal under these circumstances. Secondly, we developed a computational model for parallel execution called MHETA (*M*odel for *HET*erogeneous *A*rchitectures) that takes as input a data distribution and outputs a predicted execution time of the program. MHETA is used to evaluate distributions by both the *GBS* algorithm as well as others we used to compare its performance. This model incorporates information about the runtime environment, architecture properties, and application structure, ensuring its accuracy at runtime. These two components enable our system to determine an effective data distribution on the fly.

We determined through testing that MHETA is on average at least 97% accurate in predicting the execution times of four applications running on a variety of emulated architectures. This indicates that distributions found by search algorithms using MHETA (*GBS* included) are valid. Further, we show that our *GBS* algorithm outperforms other heuristics and is within 5% of the best distribution we found[1]. It does not scale initially as well as a greedy search algorithm, but as the number of nodes increases further, *GBS* outperforms the greedy search, which often becomes stuck at a local optimum.

1.4 Dissertation Structure

This thesis is structured as follows. Chapter 2 details related research. We introduce a summary of various important issues when scheduling parallel programs and the important of I/O awareness in these schedulers in Chapter 3. Chapter 4 continues with analysis of the proposed MPI-based runtime system to automatically solve the data distribution problem for heterogeneous architecture. Chapter 5 describes the computational model used by this system, followed by performance results for both these components in Chapter 6. We will conclude and describe some future directions in Chapter 7.

[1]The method by which we establish the best distribution used for comparison is detailed in Chapter 6

Chapter 2

Related Work

A significant amount of research has been done that is related to the work described in this dissertation. We divide it into four broad categories: modeling, data distribution, out-of-core parallel programming, and non-dedicated and heterogeneous computing. This section discusses each of these in turn and compares them to our work.

2.1 Modeling

Many of the research areas in modeling applications are summarized in a survey by [36]. One area is concerned with modeling communication, either through shared memory or explicit message passing. Another related area is in modeling the transfer of data through the memory hierarchy—from cache to main memory to disk. These models are used to either (1) establish theoretical time or space complexities, or to (2) produce performance metrics that enable researchers to compare implementations of various parallel algorithms. Therefore, they often aid in the design of efficient algorithms or prove their optimality.

2.1.1 Modeling Communication

The Block Distributed Memory (BDM) [29] model, Bulk Synchronous Parallel (BSP) [55] model and LogP [16] are three examples primarily used as a framework for designing and analyzing algorithms. The BDM model takes as input the number of processors, the message size, the maximum communication latency, and the rate a processor can inject a word onto the network. Using these inputs, the authors aim to develop algorithms that maximize computation speedup while simultaneously minimizing communication costs. The BSP model uses *supersteps*, primarily consisting of computation (and possibly I/O) and limited communication. Once a node reaches the end of the superstep after a uniform computation duration, it synchronizes with every other node. Programmers can use this model in designing algorithms, determining when to replicate versus distribute data, and deciding the optimal hardware configuration for communication. LogP models parallel programs using latency, overhead, gap, and number of processors. The computation in this model is simplified and assumed to take unit time, but LogP does consider the fact that the network has finite capacity (bandwidth). LogP was designed as a basis to develop fast, portable parallel algorithms.

Two of these models, LogP and BSP, have variants that address some of their shortcomings. LogGP [4] is an extension of LogP that was developed to more realistically calculate the overhead of long messages, which is typically subdivided and sent in pieces. Whereas LogP adds overhead to sending each piece, LogGP adds only a *single* overhead cost. Another parameter, G, is the amount of time needed to load that part of the message into the network; thus, the communication cost is the overhead plus G times the number of pieces in the message. The authors used LogGP to design and analyze algorithms, such as all-to-all remap, FFT, and radix sort. Memory logP [11] is another extension of LogP to include memory to memory communication through a distributed shared memory buffer, and it is used to predict and analyze the latency of memory copy, pack and unpack operations. Heterogeneous BSP (HBSP) [60], a variation of the BSP model, will be discussed in more detail in Section 2.4.

2.1.2 Modeling the Memory Hierarchy

Vitter and Shriver [57] developed a realistic parallel block transfer model in which a contiguous block of data is retrieved from multiple disks simultaneously. They used their model to try designing sorting algorithms that minimize the number of disk accesses they require. The Hierarchical Memory Model(HMM) [2] was used to investigate the theoretical complexity of solving problems in a machine with a memory hierarchy, particularly with exploiting locality of reference. One area of their investigation is the Least Recently Used (LRU) replacement strategy based on recent memory access patterns (on-line LRU), which moves data from smaller but expensive memory to cheaper but much larger memory. The authors used HMM to show that on-line LRU is as effective in maintaining spatial and temporal locality across levels of the memory hierarchy as the best algorithm designed using trace information.

The Parallel Memory Hierarchy (PMH) model [5] is a generic model of parallel computation that extends the memory hierarchy to multiple machines. The architecture is represented as a tree where each vertex represents a component, such as a network component or a machine's memory or disk, and the leaves represent processors. The edges are the transfer paths of data from one component to another. The PMH model includes equations for each edge to calculate latency costs and also considers contention when data from many components must share resources. They provide an algorithm to design the tree most suited to the architecture, which can be combined with measurements of specific behaviors to create an accurate model.

2.1.3 Comparison with Our Approach

Our use of MHETA is orthogonal to the goals of models like LogP. We assume that the programmer designed the best possible implementation of the parallel application, possibly by using these models described above. MHETA can then be used to determine an efficient data distribution for parallel applications during runtime. We require accurate measures of the costs to perform computation, communication and I/O for MHETA, which involve measuring these components during an instrumented run. The main focus of the distributed models is in the design of more efficient communication, thus many models make simplifications of the computation involved. The PMH model is the closest to the modeling approach we use, but unlike MHETA, it does not predict execution times, and particular architectures, such as 2-dimensional meshes, cannot be accurately modeled.

2.2 Data Distribution

There have been three primary approaches to data distribution: language annotations, compiler analysis, and run-time adaptation. We discuss them in turn.

One way to distribute data is to provide language annotations and allow the programmer to choose the distribution using application-specific knowledge. This is the approach taken by HPF [25], which was motivated by many others' work (e.g., [26]). In this approach, the programmer annotates each array, stating whether it is distributed, its alignment, and its distribution pattern. Compiler techniques to distribute (and possibly redistribute) data have also been studied extensively (e.g., [6, 24, 45, 31, 43]). The basic idea behind compiler-based systems is to analyze the source code to determine the communication pattern and then choose a `BLOCK`- or `CYCLIC`-based distribution that balances the load. Approaches employing a run-time system, such as CHAOS [28], AppLeS [7], SUIF-Adapt [38], and CRAUL [46] can use run-time information to find an efficient data distribution. This is especially effective in cases where workload and communication characteristics of a program change at run time.

2.2.1 Comparison with Our Approach

The work described above addresses the data distribution problem, not the heterogeneous data distribution problem; the latter problem allows processor speed, memory size, and I/O speed to all differ. This requires a trade-off between balancing load, minimizing communication, *and* minimizing I/O. Further, systems like AppLeS always avoid using a processor if its memory is relatively small, even if that processor could perform useful work.

2.3 Out-of-core Parallel Programming

Many researchers have studied out-of-core parallel applications, where data structures must primarily reside on disk. A programmer can write these applications just as in-core ones, using the virtual memory system with demand paging for all I/O. This approach has the advantage that the user is not required to restructure an in-core application, but it is usually inefficient because the operating system has only local information about memory access behavior. Therefore, (1) disk reads typically incur full latency and (2) the page replacement strategy can often be suboptimal for certain classes of applications. Users could alternatively manage I/O directly by inserting explicit read and write calls in their applications. Although the programmer can design file accesses with program specific patterns in mind, the restructuring of an application is error prone. Even in perfectly transformed code, disk reads still typically incur full latency, and the transfer units could be small, such as when columns are read from a matrix stored in row-major format.

Asynchronous I/O is a technique that can dramatically reduce the experienced latency of disk reads, and progress of this area is discussed in Section 2.3.1. Collective I/O techniques (in Section 2.3.2) aim to combine a series of small requests into a single large one to be handled efficiently by the file system, ensuring large transfer units and also reducing latencies. User defined virtual memory management, discussed in Section 2.3.3, allow the user to manage the virtual memory to accommodate their application's accesses patterns. Finally, work in generating efficient out-of-core applications through a combination of compiler and runtime techniques is discussed in the last section.

2.3.1 Asynchronous I/O

One extensively studied technique to hide disk read latency is prefetching. The basic idea is to separate when a disk read is issued from when the data is actually accessed. Ideally, the latency can be completely masked with overlapping computation by issuing the prefetches early enough. Issuing prefetch calls too early, however, can result in unnecessary movement of data to and from backup store as both (1) requested and (2) currently used data try to occupy memory simultaneously. This situation is difficult for out-of-core applications because the datasets currently being processed typically take up all of memory. Therefore, the system issuing prefetches needs information about an application's future access patterns, which can be determined either through compiler analysis, past behavior, or speculative execution. Mowry et al. [39] use a memory model during static analysis to establish locality of accessed variables to insert non-binding prefetches and releases when compiling an application. Agrawal et al. [3] designed a compiler that takes programs written in Fortran-D and replaces explicit regular I/O calls with their asynchronous counterparts, even across procedure boundaries. Others monitor access patterns in the operating system and use the results to prefetch data [33, 22, 61]. In [13], the authors use speculative execution while an application is blocked waiting for data to arrive from a file read to generate hints for more effective future prefetching. Finally, in [44], applications advise the OS of their future access patterns.

2.3.2 COLLECTIVE I/O

The server-directed I/O [47] and active buffering [35] in Panda, disk-directed I/O(DDIO) [32], and two-phased I/O in PASSION [10, 53], are three areas of work on *collective I/O*. All these areas of research are on distributed systems divided into two set of nodes. One set of nodes perform primarily computation, and the other set manage I/O. These two sets are called clients and servers in Panda and compute processors (CP) and I/O processors (IOP) in research in DDIO and PASSION. Any data that clients cannot store in local memory must be stored on disks on the servers. The layout of the data on the servers disks can differ greatly from its distribution onto the clients, so a client can have its required data stored in small non-contiguous pieces scattered onto many servers. The client thus will send many small I/O requests for data that is actually accessed in a regular pattern, often called the *stride*. Servers generally cannot perform any optimizations because they only have local information about how data is accessed—high-level semantic information describing the stride of reads or writes is lost. Therefore, servers tend to perform many small I/O operations for data that can actually be done with a single large operation. The basic idea of collective I/O is to introduce high-level interfaces by which this semantic information can be sent to the servers to allow it to retrieve and write data efficiently.

In the Panda library, a single client and server pair is selected as the coordinators of a collective I/O event. This client collects the read requests from the other clients into a schema detailing the local layout of the data to client memories. It then sends these schemas to the coordinating server, which shares this information to the other servers holding the pieces of data on their disks (called subchunks). For read requests, each server efficiently retrieves all the data and then sends each subchunk to the client that requires it.

In DDIO, each IOP collects read/write requests from the CPs during collective communication and determines the most efficient means to retrieve data from disk. The node responsible for the data can (1) collect several requests of data from the same disk, (2) efficiently read this single request and (3) send the results to the respective nodes. This indirection technique is most effective when the requested data is in a regular but non-sequential pattern from within the compute node and closely located items are located in different I/O nodes. This transformation also helps when the in-memory data has a different layout than on-disk data. One example occurs when a `CYCLIC` distribution is used in memory but a `BLOCK` distribution is used on disk.

2.3.3 User-Managed Virtual Memory

Research in [12] and [15] focus on the fact that the popular Least Recently Used (LRU) page replacement strategy is inadequate for a class of scientific applications with access patterns such as streaming. A tool combining the MMUM (Memory Management in User Mode) library and the MMUSSEL (Memory Management in USer SpacE Level) modules was created so the user can define his own strategy that suits the application. An alternative is to use the Transparent Parallel I/O Environment (TPIE) [56], which also hides the details of the I/O subsystem from the user. It consists of a templated library to provide efficient, high-level coordination of data movement between disk and memory. It uses the parallel block transfer model described in Section 2.1.2 in order to determine the best algorithm to move the data.

2.3.4 Compiler and Runtime Analysis

Compiling out-of-core codes is described in, for example, [52, 9]. One basic idea of this work is to extend the data parallel model to out-of-core programs. Another key idea is tiling, an optimization previously used to rewrite loops to keep data in cache as long as possible. One run-time approach, called SMARTS, schedules work dynamically to help reduce disk accesses [54]. This system employs a dataflow model to determine dependencies between various tasks to schedule work when all of their predecessors have completed. By scheduling in a LIFO manner, data in the cache produced by a predecessor is likely to still remain when accessed again. Although the focus of SMARTS was on the cache, the technique could also help out-of-core programs in which the data being read is not streamed through.

2.3.5 Comparison with Our Approach

Our work is orthogonal to techniques to improve out-of-core parallel programming. We start from an efficient user program, which can be by hand or by the above techniques. We then seek a data distribution that could potentially eliminate I/O altogether while balancing the load as much as possible.

2.4 Non-dedicated and Heterogeneous Computing

Several researchers have studied various parallel computing problems on non-dedicated or heterogeneous clusters. Gang scheduling is a technique in which multiple parallel jobs execute concurrently in a non-dedicated environment, but a subset of the processors is given exclusively to a single job at a time. Extensive research has investigated the benefits of gang versus other scheduling algorithms, such as variable and dynamic partitioning [50]. The flexibility provided by the gang scheduler makes it a competitive scheduling strategy [17]. In fact, the Lawrence Livermore National Laboratory have deployed gang scheduling onto several of its computing platforms with high success [30]. A major challenge is to determine which applications should be gang scheduled. Both message passing characteristics [49] and use of shared communication objects [19] were used to detect gang candidacy. Nikolopoulos et al. [42] also manage the trade-off between paging and gang scheduling on a multiprogrammed cluster.

There has also been work on compile and runtime libraries for running parallel applications on heterogeneous clusters. The MPI [40] and PVM [51] libraries support message passing between *different* architectures. To function on diverse architectures, such libraries must make sure, for example, to handle potentially different byte orderings. Grid MPI is an extension of MPI that enables an MPI written program to use grid services [21]. There has also been work done in compiling for heterogeneous machines [59]. This involves, for example, developing an intermediate form that permits arbitrary orderings of transformations. Finally, the HBSP model mentioned in Section 2.1.1 calculates the duration to perform computation between nodes with differing computational powers. The superstep duration is then calculated as the computation time taken by the slowest node, the slowest communication between pairs of nodes, and the communication overhead. This model can aid the user in designing an efficient parallel application that is compiled and run on a heterogeneous cluster.

2.4.1 Comparison with Our Approach

Gang scheduling is expensive to implement and is most often implemented on supercomputers. Also, [42] involves modifying the OS. Our work is focused on the runtime system on clusters, where gang scheduling is unlikely to be present. Message-passing libraries are complementary to our work, and the HBSP model does not consider the impact of I/O as MHETA does.

CHAPTER 3

I/O-AWARENESS IN GANG SCHEDULING

We first investigate the benefit of including I/O costs in scheduling distributed applications. Scientific applications running on a multiprogrammed environment often compete with other programs for resources, such as time on the CPU and memory space. Schedulers for sequential programs running on a single uni-processor machine primarily ensure that each program is allotted an equal number of time slices (a period of time to execute on the CPU). Strategies such as round robin are designed so that both large and small applications (jobs) get an opportunity to run and do not starve; this policy increases throughput as smaller jobs finish faster. A distributed application is divided into many pieces of executing instructions (processes) that run on multiple machines (nodes) and often need to communicate with each other during their execution. Therefore, schedulers for these jobs must also be able to coordinate how each process runs on its machine in the underlying architecture. Gang or co-scheduling is one such strategy, which groups the processes of an application into a gang and schedules them to run simultaneously. In doing so, it can increase the likelihood that the recipient and sender of messages at a communication point are executing at the same time, potentially greatly reducing the time a receiver of a message has to wait for the message to arrive. An effective scheduling strategy is thus vitally important to minimize a large scale scientific application's execution time.

An important input for gang schedulers is how many nodes onto which to schedule a job. One approach is to assume the job is broken into many processes and run the job on all the nodes. Unfortunately, communication costs do not scale, and the speedup for data-parallel programs eventually tails off as the number of nodes it uses increases beyond a certain point [48]. Applications with a low degree of parallelism are particularly unable to efficiently utilize large number of the additional nodes given to them. When there are too few nodes, on the other hand, the data local to a node can exceed its memory capacity, introducing I/O costs, which also adversely affects execution times. In either case, a job will take longer to finish and use resources that could have been allocated to another job, thereby increasing the completion time of both jobs.

The delay as a result of introducing I/O latencies is an order of magnitude greater than from increasing communication costs. Therefore, there is a great benefit to scheduling a job on the minimal number of nodes required to ensure that its dataset is in core (called the critical number or CN). Note that we are concerned with scheduling and not finding an efficient distribution of data. Data distribution is discussed in Chapters 4—6. Here, we assume that the job's data is divided into CN equal pieces. The scheduler can assign the processes to CN nodes and eliminate I/O while leaving nodes open for other programs to be scheduled, increasing throughput. We determined through simulation that using an application's CN would speed up its execution time by as much as a factor of three when compared to gang-schedulers that assigned all the nodes to an application. When we further modified the gang scheduler to schedule applications with smaller CN more frequently, we see an even greater throughput and reduced time to completion because more applications finished faster. I/O-awareness via a job's CN is therefore useful to bring jobs in core when possible and use the processors they were assigned most efficiently.

Section 3.1 will present a brief overview of gang scheduling, followed by details on the simulator used in Section 3.2. The last section will present results to show that the question of impact of I/O is worthy of pursuit.

3.1 Gang Scheduling Overview

Gang scheduling is an algorithm that groups all processes of a data parallel program into a gang and schedules them to run on the architecture simultaneously. Therefore, the user is presented with the illusion of a dedicated environment, and the sender and recipient processes are more likely to be executing when they reach a communication point. We assume that all these programs (*jobs*) are gang scheduled. The first subsection will describe the features of the gang scheduler used in the simulator. This is followed by special properties of gang schedulers that reduce the effectiveness of asynchronous I/O techniques.

3.1.1 The Gang Scheduler Interface

The gang scheduler interface in the simulator we designed uses a two dimensional array called a trace matrix. Although other control structures, such as the Directed Hierarchical Control (DHC)[18] have been investigated, the matrix approach was selected for its relative simplicity in implementation. Each row corresponds to a time slice, and each column represents a processor. An entry in location (i, j) contains the ID of the process to run on processor P_j when the scheduler is pointing to row R_i. When that time slice expires at a "global time slice event", the scheduler preempts and suspends all the processes running in R_i and schedules the processes in the entries located in row R_{i+1}. There is overhead in a real system during this event when coordinating all the processors, but the simulator does not measure this cost. The time slice duration is usually large enough so that the effect of this overhead is lessened.

At any time, some rows are empty, while others contain one or more non-empty entries and are considered valid. A "period" refers to the number of valid rows in the matrix. The rows are processed round robin, and when the last valid row is reached, the scheduler returns to R_0. Jobs with a predetermined number of processors are scheduled into the matrix in the first row that can accommodate them. As the active jobs increase in number and occupy more rows in the matrix, the size of the matrix also increases, and more time will elapse between successive time slices for each job. Formerly valid rows may become empty as jobs finish, so the matrix is compacted at every global time slice event to remove non-valid rows. For simplicity, the time slice duration is kept constant for all the rows in the matrix.

3.1.2 Unique Implications of Ignoring I/O in Gang Schedulers

Gang scheduled processes cannot hide I/O latencies when accessing the disk because context switches are not allowed in the middle of a time slice. CPU fragmentation [34] can result as the processors running the processes are forced to remain idle. Additionally, all processes are pre-empted when switching from one row of the trace matrix to the next, so prefetching is difficult to use. It is not possible in the general case to determine in advance if the process that issues such a request is still running when the data arrives. If in fact a different process is scheduled on the node instead, both the prefetched data from the previous process and data swapped in from disk for the current process are now loaded into memory at the same time, resulting in thrashing.

The fact that I/O latency cannot be effectively masked in general reduces the amount of progress jobs can achieve even when gang scheduling is used. The sender of a message during a communication event may be performing I/O, so the recipients of the message must contend with both the latency of the communication as well as the I/O of the sender. They will stay blocked until the sender finishes its I/O and they receive the message. In the worst case, the time slice for these processes may expire, and the sender will then only be able to send the message the next time it is scheduled. This compounded latency experienced by this job will result with it staying in the system longer and occupying rows in the trace matrix that could otherwise have been used by another job. This situation increases the turnaround time of the other jobs.

3.2 Implementation

We created an I/O-aware gang scheduling simulator by modifying the Schark simulator [8], described in Section 3.2.1. Extensions to the job description facility are described in Section 3.2.2. The primary modifications we made are encapsulated in several modules, which will be described in the last three subsections.

3.2.1 The Original Simulator

The Schark simulator is written in Java and simulates the behavior of several jobs running on a specified system. The model of computation used is probabilistic, so every feature of a job, such as execution time and number and frequency of I/O events, are based on random distributions defined

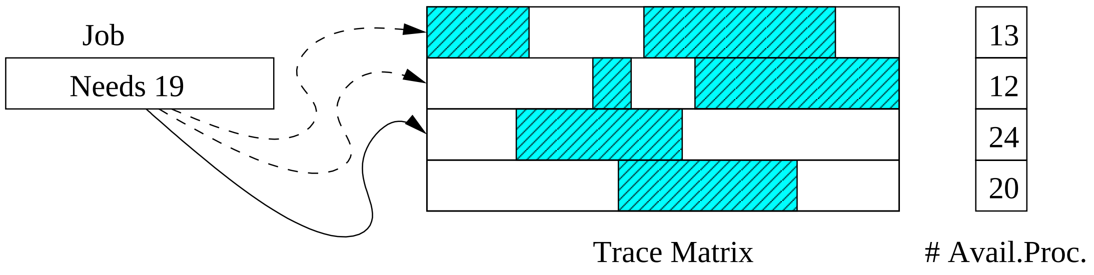


Figure 3.1: Picture of how a job is inserted into the trace matrix. The non-shaded regions of each row are unused. The job requests 19 processors, and there are 13, 12, 24, and 20 processors available in the rows of the trace matrix. The job is placed in the first row that has enough processors. In this case, it is the third row.

in a job description file. Supported distributions include uniform, hyper-Erlang, and normal. Multiple job types are allowed, and a mixture is generated when the simulator starts. The jobs are described in a *job description file*, which contains such information as I/O frequency and duration, communication frequency, initial number of processes and the time they end. It supports several kinds of interconnection networks, such as tori and meshes, the descriptions of which are included in a separate description file. Schark uses a simple gang scheduler interface and uses bin packing to schedule jobs into the trace matrix. The process IDs of a job are inserted into the first available row in the trace matrix, as shown in Figure 3.1.

3.2.2 Job Description Extensions

The introduction of I/O costs in gang scheduled jobs motivated four additions to the job description for J_i. They are: the critical number (CN_i), total sequential computation time (T_s^i), the number of iterations involved in the computation (N_i^{iters}) and the percent of its computation that can be performed in parallel (P_i). Given N_i^p, the initial number of processors allocated to the job, we can calculate the total time J_i spends working via the standard weighted average formula:

$$\text{Total Computation (work)} = \left(\frac{(100 - P_i) + (P_i / N_i^p)}{100\%} \right) * T_s^i \tag{3.1}$$

Unlike the original Schark simulator that assigned a random end time for a job, the end time now occurs when its remaining computation time is reduced to zero.

We calculate the total number of I/O events that process produces (N_i^{IO}) as a function of N_i^p, CN_i and N_i^{iters}:

$$N_i^{IO} = \begin{cases} 1 & \text{if } N_i^p \geq CN_i \\ \left\lceil \frac{CNi \cdot N_i^{iters}}{N_i^p} \right\rceil & \text{otherwise} \end{cases} \tag{3.2}$$

If $N_i^p \geq CN_i$, then the process performs only one I/O event, which is to initially bring in its working set. Otherwise, we assume that the sequential job issues I/O events CN_i times during a single iteration. The total number of I/O events is thus $CN_i \cdot N_i^{iters}$, and the parallel version is assumed to divide the I/O evenly amongst its N_i^p processors. The total number of I/O events is inversely proportional to N_i^p, so as N_i^p increases, the number of I/O events drops until it reaches one (when $N_i^p = CN_i$).

The time between I/O events $T^i_{IO \leftrightarrow IO}$ is calculated as the total sequential execution time divided by the total number of I/O events issued in the sequential program:

$$T^i_{IO \leftrightarrow IO} = \frac{T_s^i}{CN_i \cdot N_i^{iters}} \tag{3.3}$$

Finally, nearest neighbor, scatter and gather communication patterns were implemented to model three types of communication typically exhibited by distributed programs.

3.2.3 Algorithm Overview

Given J_i, the bin-packing assignment used by the original Schark simulator is simple and efficient. However, it may result in poor turnaround time and throughput in the case $N_i^p < CN_i$ as processes start to incur I/O costs. Our goal is to design a polynomial-time algorithm that runs small jobs more often in the trace matrix, while also attempting to satisfy I/O needs of a job to keep it in core. By assigning J_i more rows in the trace matrix when its critical number is less than the number of processors in the system ($CN_i < N_S^p$), it has more opportunities to run per iteration through the trace matrix. Smaller jobs therefore finish faster, potentially condensing the trace matrix as rows become empty, resulting in lower average turnaround time for every job in the system.

At a high level, the algorithm finds N_i^s rows in the trace matrix to place J_i. If the critical number for J_i is greater than N_S^p, it assigns J_i onto one row. Otherwise, N_i^s is calculated as the ratio of N_S^p and CN_i.

$$N_i^s = \begin{cases} \lceil N_S^p / CN_i \rceil & \text{if } CN_i \leq N_S^p \\ 1 & \text{otherwise} \end{cases} \tag{3.4}$$

Accordingly, we assign J_i its critical number of processors unless that exceeds N_S^p:

$$N_i^p = \begin{cases} CN_i & \text{if } CN_i \leq N_S^p \\ N_S^p & \text{otherwise} \end{cases} \tag{3.5}$$

It then finds N_i^s rows in the trace matrix that can accommodate J_i.

We also take into account not only the entering job, but all the other jobs in the system. The next section describes our algorithm in detail.

3.2.4 Algorithm Details

This section formalizes the algorithm used to schedule jobs and is split into two portions. The first focuses on the fitness of row and processor assignments of job J_i in isolation, and the second calculates the overall fitness of all the jobs. Those with lower fitness spend less time performing computation, so their progress is slowed. Our ultimate goal is to select a configuration for J_i that results in the most system-wide progress to job completion. Although the description included here is based on the features of a particular job, the times are actually measured at the level of its processes. Once these numbers are known, they are integrated to model the execution behavior of the job itself.

We first define our assumptions, followed by a description of several additional variables we use in this function. The third subsection describes the function, and we conclude with the time complexity analysis to compute the entire fitness function.

ASSUMPTIONS AND VARIABLE DEFINITIONS

We make a few assumptions to simplify the design of the fitness function. First, N_i^p and N_i^s are write-once variables. If they are allowed to change, extensive detection is required. Our algorithm is designed to schedule J_i into multiple rows of the trace matrix, and there is no guarantee that a particular process will be assigned to the same processor for each row. Therefore, a process will likely need to migrate (i.e. move from one processor in one row to another processor in another row). By assuming the architecture is homogeneous, the cost for migration can be calculated in the same manner regardless of the direction of movement. Finally, communication costs are not factored into the fitness function (but are in job execution), as they complicate the model but are often dominated by I/O latency in out-of-core parallel programs.

There are several variables that need to be defined for this function, and they will also be used in the remainder of the chapter. Four variables deal with times spent performing I/O, context switches and migration. The value in T_w^β is the amount of time required to load the next required data elements from disk during a *single* explicit I/O event. This value is a function of the size of the data accessed. The variable T_w^i denotes the total amount of time job J_i spends loading the working set during one trace matrix iteration. Variables T_c^i denotes the total context switch time, and T_m^j denotes the total migration time between different processors for J_i on one trace matrix iteration. A fifth variable, T^s, stores the duration of a time slice in the system.

There are five other variables to store various values of some of the characteristics of the system or of J_i. Three variables, N_S^p, N_i^p, N_i^s and have already been defined. The variable N_M^r is the number of rows in the trace matrix after J_i is inserted. Finally, N_i^{IO} stores the number of I/O requests performed by J_i per time slice. This value can be calculated by multiplying the I/O frequency by T^s. For k jobs, there have to be k copies of each of these three variables. Table 3.1 summarizes these variables.

Name	Description
T_w^β	amount of time required to load the required data from disk during an explicit I/O event.
T_w^i	total amount of time per iteration of the trace matrix J_i spends performing explicit I/O
T_c^i	total amount of time per iteration of the trace matrix J_i spends in a context switch.
T_m^i	total time that J_i spends migrating during a single iteration through the trace matrix.
T^s	duration of a time slice in the system.
N_S^p	total number of processors in the system.
N_i^p	number of processors alloted to this job.
N_i^s	number of time slices for J_i.
N_M^r	number of rows used in the trace matrix after the last job, J_{last}, is inserted.
N_i^{IO}	number of I/O requests performed by this job per time slice.

Table 3.1: Variables Used in Implementation of the Fitness Function

PROGRESS MEASUREMENT

The optimum progress a job J_i can make towards completion during a single iteration through the trace matrix is:

$$T_o^i = (N_i^s * T^s)$$

The actual progress is limited by time spent performing I/O, time spent context switching, and time spent in migration. The last two factors are the result of assigning J_i to multiple rows, and our fitness function must include these. The first and most significant factor is T_w^i, the total amount of time required to move data between disk and memory during a single iteration, which increases with the number of disk accesses. This makes it vital to keep a job in core, if possible. In our model, we assume that the processes of a job all have the same number of I/O requests. Hence, T_w^i is equal to the product of (1) the number of I/O requests issued during a time slice, (2) the number of time slices allocated to J_i, and (3) the duration to retrieve the data:

$$T_w^i = N_i^{IO} \cdot N_i^s \cdot T_w^\beta$$

The second factor is the cost associated with context switching the processes of J_i when its current time slice starts. When necessary, it consists of the duration to restore the state of a process as well as loading its working set from swap. Suppose J_i is assigned to only two rows in the trace matrix (R_a and $R_{a'}$). A process scheduled on the j^{th} processor has no context switch costs if it is the only one running on that processor between R_a and $R_{a'}$. Otherwise, the process incurs overhead to restore its state and load its working set that had been swapped out when a process of a different job ran on that processor. We calculate T_c^i as the average of all context switch costs incurred by all of its processes during a single iteration through the trace matrix.

The third factor to reduce T_o^i is the cost, if any, for the processes of J_i to migrate between processors. This only occurs when a process runs on one processor in row R_a and a different one in $R_{a'}$. If the processes stay on the same processors, this duration is reduced to zero. The value of T_m^i is also the average migration time over all of the processes in J_i during one iteration in the trace matrix.

The total amount of progress J_i makes towards completion during one iteration through the trace matrix is the optimum minus the three factors discussed above (time spent accessing the disk, context switching, and migrating):

$$T_\rho^i = T_o^i - \left(T_c^i + T_m^i + T_w^i\right)$$

Intuitively, the relative progress of any job in the trace matrix depends on N_M^r, which may increase as a result of adding one or more rows to the end in order to fit J_i. We can calculate the total amount of relative progress of all the jobs during one iteration of the trace matrix (T_ρ^C) as the sum of the relative progress of all the jobs.

$$T_\rho^C = \frac{1}{N_M^r} \cdot \sum_{i=0}^{k} T_\rho^i$$

TIME COMPLEXITY OF FITNESS FUNCTION

For each job J_i the time complexities to evaluate the time all k jobs spend during explicit I/O, context switch and migration are $O(1)$, $O(N_M^r \cdot N_i^p \cdot N_i^s)$ and $O(N_i^s \cdot N_i^p)$ respectively. It is clear that $(\max_{i=0\dots k} N_i^s) \leq N_M^r$, so the time complexity to perform the fitness function (T_f) for all k jobs is

$$T_f = O\left(\sum_{i=1}^{k} \max\{O(1) + O(N_M^r \cdot N_i^p \cdot N_i^s) + O(N_i^s \cdot N_i^p)\}\right) = O\left(k \cdot (N_M^r)^2 \cdot N_i^p\right)$$

which is polynomially bound to the number of jobs in the system, the number of slices for J_i, the number of rows the matrix, and the number of processors for J_i.

3.2.5 MULTI-ROW SCHEDULING ALGORITHM

The next major modification involves generating the configurations for scheduling J_i; each is evaluated using the fitness function, and the best is chosen. The goal here is to maximize the overall progress of *all* the jobs for a single iteration based on the principle that this measure leads to best average turnaround time.

The number of possible configurations to generate and evaluate its "fitness" is the same as an N_i^p-combination of the set of processors in the system. The size of this set is N_S^p, so the number of configurations is "N_S^p choose N_i^p" with the following complexity [14]:

$$\begin{aligned}\begin{pmatrix} N_S^p \\ N_i^p \end{pmatrix} &= \frac{N_S^p!}{N_i^p! \cdot (N_S^p - N_i^p)!} \\ &= \frac{N_S^p(N_S^p-1)\cdots(N_S^p - N_i^p + 1)}{N_i^p(N_i^p-1)\cdots 1} \\ &= \left(\frac{N_S^p}{N_i^p}\right)\left(\frac{N_S^p-1}{N_i^p-1}\right)\cdots\left(\frac{N_S^p - N_i^p + 1}{1}\right) \\ &\geq \left(\frac{N_S^p}{N_i^p}\right)^{N_i^p}\end{aligned}$$

which is exponential in N_i^p. It is computationally intractable to generate all possibilities (particularly when $N_i^p = N_S^p/2$), so we designed a pruned, polynomially-bound search algorithm to reduce the complexity. This algorithm generates an initial set of processors for a target row from the trace matrix, and then selects processors from the other possible rows that match these as closely as possible. We first describe the input to the algorithm, followed by details of the principles used to guide the selection of processors and rows.

INPUT

The input to this algorithm is a condensed version of the trace matrix that has only the “useful” rows—those that have N_i^p processors free. (Any other rows cannot possibly be assigned.) This information is generated by another function and is stored in a set of two vectors and two matrices, one set for processor information and another for row information. The vectors A^p and A^r store the number of processors that are free in the row corresponding to position A_i^p and the number of rows in which the j^{th} processor occurs, respectively. The matrix D^p stores the indexes of the processors unused in these rows. The matrix D^r stores the indexes of the rows the j^{th} processor is unused. These two variables have back-mapping vectors B^r and B^p to refer to the row and processor index corresponding to the positions in the first two vectors. Figure 3.2 shows an example of the relationship between two rows in D^p and the trace matrix.

GUIDING PROCESSOR CHOICES

The first step involves generating a combination of processors for the target row (R_t) used later. Intuitively, a processor free in many useful rows is likely to result in an efficient configuration because migration occurs less frequently. Accordingly, we rank the processors in decreasing order based on how many rows they occupy, which is stored in A^r. If all processors are considered, the algorithm is intractable. However, if only the first N_i^p processors are then selected, there is no allowance for variability, and a potentially good choice may be pruned prematurely. Let the variable α, which we pass as an algorithm parameter, be the number of additional processors that are included beyond the first N_i^p processors. As α decreases, the value is reduced, and the complexity on the number of combinations also decreases. However, a value that is too low may eliminate too

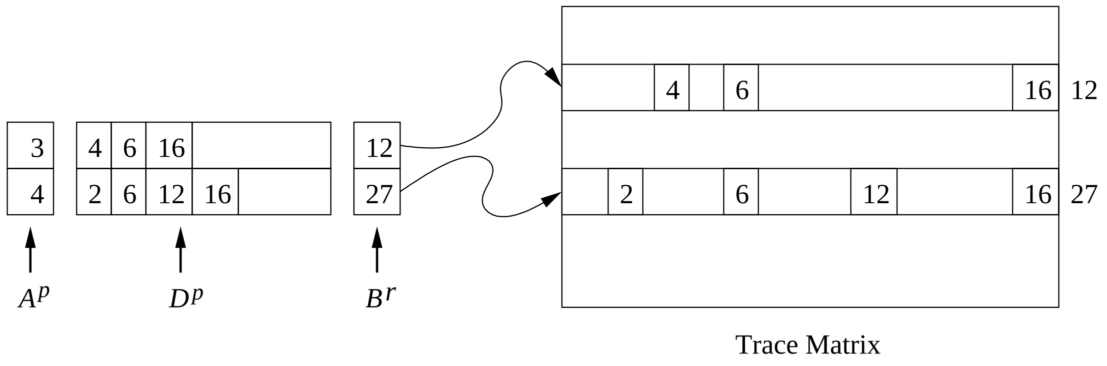


Figure 3.2: Picture of the relationship between A^p, D^p, B^r and the trace matrix. The job J_i needs 3 processors from two rows in the trace matrix. Only rows R_{12} and R_{27} contain enough processors, and the rest are ignored. The processor indices P_4, P_6 and P_{16} from R_{12} are stored in D_1^p and the indices P_2, P_6, P_{12} and P_{16} from R_{12} in D_2^p. A_1^p contains 3 because there were three processors from R_{12}, and A_2^p contains the number of processors indices were used from R_{27}.

many potentially good combinations from consideration. Currently, we set $\alpha = 2$. The processors not included in the top $N_i^p + \alpha$ are removed from D^p. Note that removing a processor from D_k^p may potentially result in $A_k^p < N_i^p$, whereupon that entire row is also removed from D^r.

Out of these $N_i^p + \alpha$ possible processors, combinations of N_i^p are selected for R_t. There are $O\left((N_i^p + 2)^2\right) = O\left((N_i^p)^2\right)$ such combinations. The next task is to find $N_i^s - 1$ rows from the remaining "useful" rows with processors that closely match the combination of processors in R_t (to avoid excessive migration).

Guiding Processor Matches

The basic idea is that given a selection of processors for R_t, the combination of processors selected from any other row in D^p is matched as closely as possible to R_t. Due to the sorting and pruning process described in the previous section, the value of α already limits how different the processors can be between these two rows.

The algorithm starts with R_t and performs an initial scan through the remaining rows in D^p. For each row R_j, all the processors in it are compared with the processors in R_t. All the matches are saved in a vector M, and all the other processors are saved as alternatives in a vector W. If after the comparison with R_j, N_i^p matching processors were found, this row and processor information is saved as part of the configuration for J_i. Otherwise, the processors from W are combined with the contents of M and saved as an alternative row.

If at the end of this scan there are N_i^s rows with an exact match, the algorithm immediately tests the configuration using the fitness function. Otherwise, there are $x < N_i^s$ such rows, and the alternative rows generated during the initial scan are sorted in decreasing order based on how many of their processors match with those in R_t. The best $N_i^s - x$ are selected and included in the configuration. Then the complete configuration is tested using the fitness function. The configuration with the best fitness measure is the one that the scheduler uses when scheduling J_i.

This strategy enables very controlled migration because any task that migrates from its processor in row R_t to a different one in another row must return to its original processor when the scheduler returns to R_t. The simulator was implemented such that any tasks that do not need to migrate stay on the same processor. This decision limits overhead and makes the encoding more elegant.

Time Complexity for the Complete Algorithm

The target row R_t is selected from D^p at most N_M^r times, each with $O\left((N_i^p)^2\right)$ processor combinations. There are $O\left(2N_S^p\right)$ comparisons between each R_t and at most N_M^r other rows. Any alternative rows are sorted in $O\left((N_M^r)^2\right)$ steps after all of the processor comparisons. At each combination, the configuration is evaluated using the fitness function, so the time complexity of the complete algorithm is:

$$O\left(N_M^r \cdot (N_i^p)^2 \cdot \left((N_M^r \cdot 2N_S^p) + (N_M^r)^2\right) \cdot T_f\right) = O\left((N_M^r)^2 \cdot (N_i^p)^2 \cdot (N_S^p + N_M^r) \cdot T_f\right)$$

This algorithm is therefore polynomially bound to the number of jobs in the system, the number of rows in the trace matrix and the number of processors for J_i and in the system.

3.3 Gang Scheduling Performance

We ran the modified Schark simulator on several mixtures of jobs to test the effectiveness of I/O awareness on performance. The simulated architecture was a mesh with 128 nodes, and the hard time limit was set to 80,000 simulated time units (STU), a sufficiently high number so that all of the jobs finished. There were 100 jobs, where each had 200 STU of computation to finish, and all of them were introduced at startup. (Separate tests with jobs arriving uniformly produced largely similar results.) Each time slice had the duration of 20 STU. The jobs did not perform any synchronization apart from the communication events. All of the jobs cycled through a pattern of nearest-neighbor, gather and scatter types of communication. The duration to access the disk was set to two STU. The value of N_i^{iters}, the number of iterations the job cycles through, was set to 100 for the first set of tests, but was modified later to simulate different I/O frequencies, as will be described in Section 3.3.3. Our parameter settings were loosely based on our experience implementing an out-of-core iterative PDE solver on an IBM SP2.

We produced and introduced a variety of jobs into the system by setting the values of two key aspects over a uniform distribution: the percent of the job's computation that can be performed in parallel (P_i) and the job's critical number (CN_i). P_i ranged over $(65, 90)$, and the baseline CN_i was sampled over the range of $(1, N_S^p)$ and rounded up to a power of 2. This range was modified in later tests as will be described in Section 3.3.2. The time between communication events ranged over $(0.2, 0.3)$. This was the amount of time the job spent in computation between communication points. Table 3.2 summarizes this information.

The simulator was run in three stages to test the effectiveness of I/O awareness, and also how it handled job mixtures in which the I/O frequency and critical number ranges varied. The first stage, described in Section 3.3.1, kept the number of I/O requests and the critical number ranges constant while the number of tasks for each job varied. This stage established the baseline results and basic framework for the tests in the remaining stages. The second stage varied the critical number range,

Field Name	**Values**
Maximum Simulator Runtime	80,000 STU
Number of Nodes	128
Architecture	Mesh
Job Interarrival Time	0 STU (At startup)
Number of Jobs	100
Required Computation Time	400 STU
Percent Parallel	Unform Distribution over range (65,90)
Communication Pattern	Nearest Neighbor, Gather, Scatter
Communication Frequency	Uniform Distribution over range (0.2,0.3)
I/O Duration	2 STU
Critical Number	Uniform Distribution over $(1, N_S^p)$ (baseline)

Table 3.2: Common Field values of generated jobs. (Note: STU = Simulated Time Units)

and Section 3.3.2 describes the results from this change. The third stage only varied the number of I/O requests generated by every job, and its results are discussed in the last section.

We include both the turnaround times and throughput in our results. The latter is measured via the simulator run time because the number of jobs in the system is constant at 100, and all the jobs are introduced at startup. In general, good turnaround time is correlated with good throughput, but there are some cases where we were able to optimize for turnaround time and tolerate poorer throughput.

3.3.1 Baseline Test

The basic structure of each set of tests measures performance in two respects. We examine the impact of enabling versus disabling the multi-row scheduling mechanism we developed. When disabled, the simulator is operating under what we call the “Schark” mode because the scheduling algorithm is similar to bin packing used by the unmodified gang scheduling interface. Otherwise, the simulator is in “Enhanced” mode. Note that if all 100 jobs are assigned all the processors, each job will be assigned only one row of the trace matrix, so whether or not the multi-row algorithm is used does not impact the results. For a given run of the simulator, the mode is fixed in either "Schark" or "Enhanced" mode.

We also measure the impact of the four different ways we calculate the number of processors, N_i^p, each job J_i entering the system gets. N_i^p is computed using one of the methods below before the job is actually scheduled.

1. "c-based": N_i^p is set as detailed by Equation 3.5 in Section 3.2.3. We compare the impact of the "c-based" method with the other three in the analysis.

2. "p-based": an inherently conservative method intended to model the current state of gang scheduling where the user estimates N_i^p probably based on speedup, without considering I/O costs. N_i^p starts with the value 1 and is successively doubled. At each value of N_i^p, the computation time for the job is calculated using Equation 3.1. Ideally, as the number of processors is doubled, the speedup should also double. The speedup of 1.2 was selected as the point at which the speedup was no longer large enough to justify the increase in processors. When the speedup resulting from doubling N_i^p drops below this threshold, the value of N_i^p is set to be the number of processors this job had before this drop occurred. If doubling N_i^p reaches N_S^p, the job receives all processors.

3. "a-based": set N_i^p to N_S^p, the number of processors in the system, regardless of the job's critical number or I/O requirements. This method simulates the situation where greedy users decide to use all the resources available. Note that this method does not consider the job's speedup behavior, so efficiency is ignored.

4. "cP-based": a hybrid of the other methods. First, N_i^p is computed using the "p-based" computation. If after this step $N_i^p < CN_i$, $N_i^p \leftarrow N_S^p$. This can model the situation where both greedy and conservative users have submitted their jobs into the system.

The results show that for a specific critical number and I/O frequency range, the simulator that ran in Enhanced mode and used the "c-based" method of task computation had the best overall turnaround time. The “p-based” method underestimated a job’s critical number, which resulted in many jobs that issued many I/O requests. The jobs that had their N_i^p calculated by the “c-based” method issued on average only one I/O request because their critical number was at most N_S^p. Therefore, the jobs in the latter case suffered the 2 STU penalty only once, so the average turnaround time when in Schark mode is a factor of 5.2 faster than the latter case in which the jobs incurred the penalty multiple times. Further, the Enhanced mode allowed for smaller jobs to finish faster, so in this mode the “c-based” method is slightly faster, 5.4 times better than the “p-based” method. The “c-based” method is 31% faster than the “a-based” method in Schark mode and 42% faster when in Enhanced mode. This result is due to the fact that in the latter case there were more jobs that were allocated more than their critical number of processors and were unable to efficiently use them. The results of using the “c-based” method is 25% faster in Schark mode and 30% faster when in Enhanced mode when compared to the results of the “cP-based” method. This result is expected because some of the jobs did manage to get their critical number of processors, while others were assigned too many. The “c-based” method also establishes the lower bound of N_i^p, so the scheduler could pack the jobs more tightly into the matrix than either the “cP-based” or “a-based” methods while keeping them in core. A more compact usually results in a faster turnaround time because the time to proceed through one iteration of the matrix is faster. Table 3.3 summarizes these results.

3.3.2 Changing the Critical Number Range

The second stage ran the simulator with three different ranges for CN_i: $(1, \frac{1}{2}N_S^p)$, $(1, N_S^p)$ and $(1, 2N_S^p)$. As in the baseline test, both Schark and Enhanced mode was used and N_i^p was varied. The results show that the “c-based” method with the simulator in Enhanced mode produced the fastest turnaround for when the critical number range was both halved and doubled.

When the critical number range was halved, the “c-based” is a factor of 4.2 and 3.9 better than the “p-based” method in Schark and Enhanced modes, respectively. These improvements are less pronounced than in the baseline tests because the “p-based” method produced smaller values

I/O mode	TCM	ATT	End Time
Schark	p-based	15912	40007
Schark	a-based	4455	5912
Schark	cP-based	4109	5513
Schark	c-based	3074	4302
Enhanced	p-based	14132	22666
Enhanced	cP-based	3706	5532
Enhanced	c-based	2597	4792

Table 3.3: Results in average turnaround time (ATT) and end times of various task compute methods (TCM) from baseline tests in STUs.

for N_i^p, which was closer to CN_i. Hence, more jobs were actually given their critical number of processors. The “c-based” method was, however, 2.4 and 2.5 times better than the “a-based” method when the simulator was in Schark mode and Enhanced mode, respectively. This result is due to the fact that every job’s critical number was at most half of N_S^p, so assigning all the processors to them created inefficiencies that were even more pronounced than in the baseline test. Comparing “c-based” with “cP-based” methods also results in a more pronounced improvement in average turnaround time for the same reason. Decreasing the critical number for the jobs in the system therefore can increase in general the effectiveness of the I/O-aware scheduler because allocating a job’s critical number of processors to it can result in a more efficient execution and faster turnaround times.

When the range for the critical number extended to twice the number of processors in the system, the results show that the simulator with the “c-based” processor computation method fared only about a 4% and 9% better than the “a-based” method in Schark and Enhanced modes. The improvement “c-based” method has over the “cP-based” method was 3% in Schark and 7% in Enhanced mode. These results, which are worse than the baseline case, is due to the fact that the critical numbers were so high that the “cP-based” method was more accurate in estimating CN_i. The “c-based” method still more than halved the average turnaround time compared to the

CN Range	I/O Mode	TCM	ATT	End Time
Half	Schark	p-based	8007	20464
Half	Schark	a-based	4455	5912
Half	Schark	cP-based	4036	5260
Half	Schark	c-based	1864	2682
Half	Enhanced	p-based	6976	12072
Half	Enhanced	cP-based	3416	5392
Half	Enhanced	c-based	1752	3642
Twice	Schark	p-based	31708	79124
Twice	Schark	a-based	13305	24170
Twice	Schark	cP-based	13179	23908
Twice	Schark	c-based	12726	23388
Twice	Enhanced	p-based	28021	43877
Twice	Enhanced	cP-based	13033	23990
Twice	Enhanced	c-based	12123	23608

Table 3.4: Results in average turnaround time (ATT) and end times of various task compute methods (TCM) from varying the Critical Number Ranges in STUs.

"p-based" method when the simulator was in Enhanced mode, which suggests that the penalty for issuing an I/O request is greater than the lack of speedup. The drop was even more dramatic when in Schark mode, which is likely due to the fact that employing the Enhanced mode reduced the average turnaround time for the "p-based" method. These results indicate that changing the job mixture such that an increased number of jobs had critical numbers larger than the number of processors in the system reduces the effectiveness of I/O awareness because more jobs will be given all the processors in the system regardless of whether the "a-based", "c-based", or "cP-based" methods are used. However, I/O-aware schedulers still outperform those that use a "p-based" algorithm because there is an increase in the number of I/O events that are underestimated. Table 3.4 summarizes these results.

3.3.3 Changing the Frequency of I/O

The simulator was run again but with first halving, then doubling, the I/O frequency while keeping the range of critical numbers at $(1, N_S^p)$. The first tests halved the value of N_i^{iters} for each job, which halved the I/O frequency using the equation in Section 3.2.2. The simulator in Enhanced mode with the "c-based" method again results in the fastest turnaround time, but it is only 2.7 times faster than the "p-based" approach in either Schark or Enhanced mode. This result is expected because the total number of I/O requests was halved, so the total penalty incurred was also reduced. It is 31% and 64% better than the "a-based" method and 25% and 30% better than "cP-based" method for the simulator in Schark mode and Enhanced mode respectively. Both sets of results are primarily based on the effect of speedup because the jobs in all these cases issued only one I/O request. However, the scheduler was still able to determine the lower bound on the value of N_i^p and hence was able to assign fewer than N_S^p processors to some jobs. This could enable it to compact the trace matrix and shorten the elapsed time between points at which the jobs were scheduled, thereby reducing turnaround time for the former set of results.

As expected, doubling the I/O frequency, by doubling N_i^{iters}, in the second set of tests enhanced the benefits of I/O awareness when comparing the "c-based" and "p-based" methods. The "c-based" method was 10.12 and 10.64 times faster than the "p-based" method when the simulator was in Schark and Enhanced mode, respectively. This is due to the significant number of I/O requests issued by the jobs that did not receive the critical number of processors. The turnaround times for the "a-based" and "cP-based" methods are very similar to their times in the first set of tests in which the file frequency was halved. $CN_i \geq N_S^p$ for all jobs, so they are in core. Changing the I/O frequency does not affect jobs that only issue one I/O request. The ratio of improvement between these two methods and "c-based" is thus about the same. The results are summarized in Table 3.5.

I/O Factor	I/O Mode	TCM	ATT	End Time
Half	Schark	p-based	8277	20484
Half	Schark	a-based	4455	5912
Half	Schark	cP-based	4109	5515
Half	Schark	c-based	3079	4317
Half	Enhanced	p-based	7177	12066
Half	Enhanced	cP-based	3705	5532
Half	Enhanced	c-based	2603	4812
Double	Schark	p-based	31191	79152
Double	Schark	a-based	4455	5912
Double	Schark	cP-based	4089	5493
Double	Schark	c-based	3082	4338
Double	Enhanced	p-based	27597	43878
Double	Enhanced	cP-based	3706	5532
Double	Enhanced	c-based	2597	4792

Table 3.5: Results in average turnaround time (ATT) and end times of various task compute methods (TCM) from varying the I/O frequencies in STU.

3.4 CONCLUSION

I/O awareness in gang schedulers can result in an enormous improvement in turnaround time for jobs in a system when compared to systems that ignore the I/O requirements of its job mix. Enhancing the scheduler with an algorithm to schedule jobs that span multiple rows based on the relationship between its critical number and the total number of processors in the system can further increase the likelihood that they will be able to finish faster. We have used a simulated gang scheduler that verified that such results are possible for a variety of jobs introduced into a system.

CHAPTER 4

I/O COSTS IN SEARCHING FOR A DATA DISTRIBUTION

The previous chapter has shown the impact of I/O in scheduling out-of-core applications on a homogeneous cluster. This chapter changes the focus to its impact on a different problem—finding a data distribution that minimizes application execution time. Previous research on this problem primarily focused on balancing the load of in-core applications running on a homogeneous cluster. My research extends this problem to include *heterogeneous* architectures (a cluster of machines with differing characteristics), which increases the likelihood that applications are out of core—and hence I/O occurs. For a runtime system to solve the out-of-core data distribution problem, two important components are necessary. The system first requires a mechanism that searches through possible distributions, which will be described in this chapter. This search algorithm uses a computational model that accurately evaluates each candidate distribution, which will be described in Chapter 5.

The solution to the data distribution problem must be found at runtime, so an alternative to the computationally intractable brute-force search must be used. The search space can be greatly reduced by assuming distributions along a single dimension, but the search space complexity is still exponential, as shown in Section 4.1. We designed two algorithms that finish in a polynomially-bound number of steps given monotonicity between a data distribution and the application execution time. The first, called Generalized Binary Search (*GBS*), is based on binary search and is optimal under the monotonicity assumption but relatively slow. The second is based on greedy search (*GS*), which is faster than *GBS*, but can get stuck in a local optimum. Section 4.2 formalizes the problem and proves the optimality of the *GBS* algorithm. Section 4.3 details two other search heuristics, a genetic algorithm (*GA*) and simulated annealing (*SA*), that we later use to compare

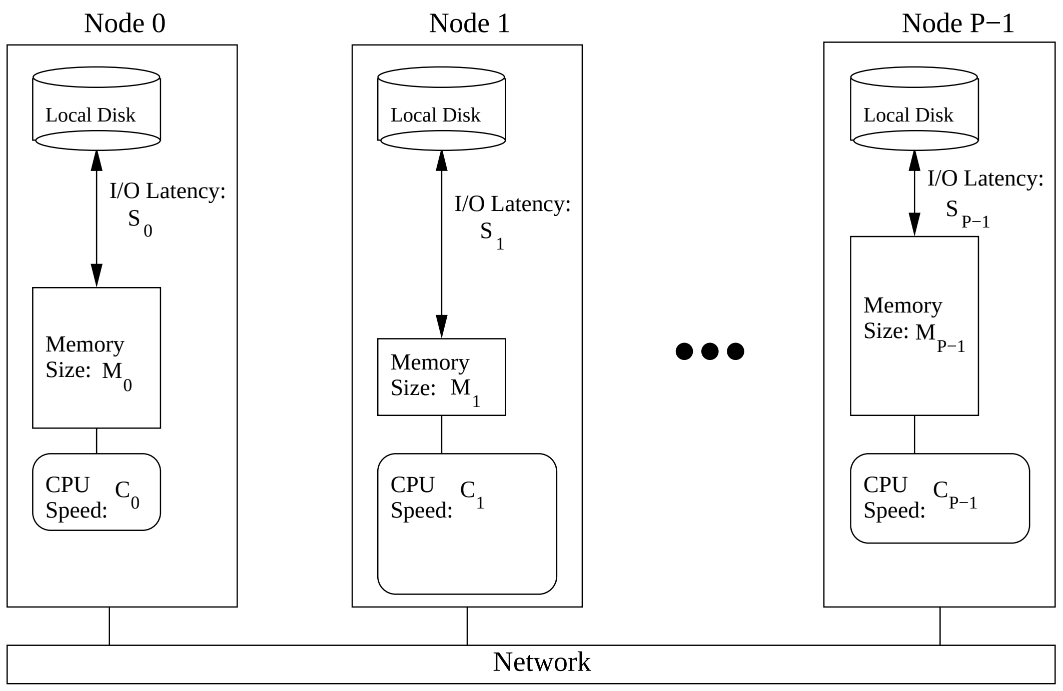


Figure 4.1: The composition of the heterogeneous architecture emulated for our experiments. Note that the memory capacities, I/O speeds, and processor speeds between any two machines can differ, represented by different sized boxes in the figure. The sizes of the boxes in the figure for memory and CPUs represent relative storage capacity and CPU power.

the performance of *GBS* and *GS*. (The relative performance of all four algorithms are detailed in Chapter 6.)

4.1 Problem Complexity

Figure 4.1 displays a typical heterogeneous cluster consisting of P nodes, numbered 0 to $P-1$. We assume that each node has a local disk and any communication is achieved through message passing over the network. The relative CPU power of each node is based on its processor speed; i.e., the relative rate at which it can process elements in the dataset. The nodes have differing amounts of available physical memory and I/O latencies to read and write data from local disk. Note we assume a commodity cluster, as opposed to a much more costly RAID system or global disk used by all the processors.

The data distribution problem for such a cluster is strictly harder than the homogeneous, in-core data distribution problem—which is itself NP-complete [23]. Our runtime system currently seeks a distribution in only one dimension, but even this problem is computationally intractable, as shown by the following theorem.

Theorem 1. *Suppose that S is the size of the dimension distributed on P processors. Consider a lattice of all non-negative coordinates in P-dimensional space. The exhaustive algorithm of trying* all *points in P-space whose sum is S will be order of magnitude S^{P-1}.*

Proof. By induction. In the base case with 2 dimensions (when $P = 2$) we have exactly $S + 1$ points $(0, S), (1, S - 1), \ldots, (S, 0)$, each of whose components sum to S. Assume the induction hypothesis for P dimensions and consider $P + 1$ dimensions. For each first coordinate value x_1 (whose value is between 0 and S), the sum of the other P coordinates must be $S - x_1$. By the inductive hypothesis, the number of such points is the order of magnitude of $(S - x_1)^{P-1}$, which is less than or equal to S^{P-1}. If we sum this over x_1 from 0 to S, the result is an order of magnitude $\leq (S + 1)S^{P-1}$, which is order of magnitude S^P. $\square$

Clearly, exhaustive testing is not feasible, given that S is large, and we want P to be able to scale up to large values.

4.2 Problem Formalization

Assume a heterogeneous cluster as shown in Figure 4.1. Let V be the set of vectors with P components, where each component is a nonnegative integer (i.e., $x_i \in \mathbb{N}$). Our problem is for all vectors $x \in V$, to determine an algorithm for approximating the solution of

$$\min_x(\max_i(f_0(x_0), \ldots, f_i(x_i), \ldots, f_{P-1}(x_{P-1}))),$$

where

$$f_i(x_i) = g_i(x_i) + h_i(x_i) + r_i(x_i),$$

subject to the components of x totals to S:

$$\sum_{i=0}^{P-1} x_i = S, \quad x_i \in \mathbb{N}.$$

Functions $g_i(x_i)$, $h_i(x_i)$, and $r_i(x_i)$ represent the computation, I/O, and communication time on the i^{th} node, respectively. Function $g_i(x_i) = c_i x_i$, where all $c_i > 0$. Function $h_i(x_i)$ is a step function that has steps N_i, $2N_i$, $3N_i$, $\ldots$, of size $2k_i$, $3k_i$, $4k_i$, $\ldots$, each positive. (The steps begin at $2k_i$ because if any reads from disk are required for a given array, there must be at least *two* accesses to disk. If only one read is necessary, it can be performed once at the start of execution, and the application is in core.) Clearly $g_i(x_i)$ and $h_i(x_i)$ are both monotonically non-decreasing functions, while $r_i(x_i)$ is more complicated because communication involves not just send or receive overheads (as in the LogP model[16]). The nodes also block when a node executes a blocking receive—or a wait operation, if receive is asynchronous—before the arrival of the desired data. The MHETA model (see Section 5.3) can predict both (1) how long each node spends executing without considering blocked time and (2) the total execution time on all the nodes with blocking. We assume for our *GBS* and *GS* search algorithms that $r_i(x_i)$ consists only of communication overhead (o_i), thus making it and $f_i(x_i)$ monotonically non-decreasing functions. In support of this assumption, we note that for the communication patterns we support (nearest-neighbor and reduction), communication is mostly uniform between nodes. This means that if the computation and I/O are balanced between nodes, a near-optimal distribution can be found.

We now present a theorem and lemma that are later used to show that the *GBS* algorithm is optimal:

Theorem 2. *Let x and x' be* ***any*** *two vectors in P space. Then,*

$$\max_i(f_i(x_i)) \geq \min_i(f_i(x'_i))$$

Proof. By contradiction. Suppose it is not true and $\max_i(f_i(x_i)) < \min_i(f_i(x'_i))$. Then for each j^{th} component $f_j(x_j) < f_j(x'_j)$ because $f_j(x_j) < \max_i(f_i(x_i)) < \min_i(f_i(x'_i)) < f_j(x'_j)$. However, each f_i is monotonically non-decreasing, and as $\sum x_i = \sum x'_i = S$, at least one $x_i \geq x'_i$, so $f_i(x_i) \geq f_i(x'_i)$. $\square$

We can reason that the solution for a vector x is thus bounded by the maximum and minimum f-values:

Lemma 1. *For a given vector x, let $M = \max_i(f_i(x_i))$ and $m = \min_i(f_i(x_i))$. Then $m \leq \min_x(\max_i(f_i(x_i))) \leq M$.*

Proof. Clearly $m \leq M$. If $\min_x(\max_i(f_i(x_i))) < m \leq M$, then certainly the x' where the minimum occurs would have all components $< m$. In particular, due to monotonicity, all $x'_i \leq x_i$ with at least one $<$. But $\sum x'_i = \sum x_i = S$, so this is impossible. $\square$

Observation 1. *Given $\delta > 0$ (not necessarily small), if we can find an $x \in V$, where $\sum x_i = S$ with $M - m < \delta$, then we have approximated the min max to within δ.*

Note that this observation enables us to bound the optimal solution *both* from above *and* below.

4.3 The Search Algorithms

This section presents the different search algorithms we implemented to find an effective data distribution: a generalized binary search (*GBS*) and greedy search (*GS*) algorithms we designed, a genetic algorithm (*GA*), and simulated annealing (*SA*). Note that each requires an evaluation function for each candidate data distribution, and we chose the application's execution time as a result of that distribution as a measure of its "desirability". To actually run the application to determine its execution time is impractical, as these algorithms must search through *many* distributions at runtime. We therefore use MHETA to estimate its execution times, which is described in Section 5.3.

4.3.1 Generalized Binary Search

The *GBS* algorithm exploits the monotonicity of the evaluation function f, that is, with the simplified $r_i(x_i)$. From a randomly chosen vector $x \in V$, we improve the application completion time by continuously reducing the largest value of $f_i(x_i)$ over $x_i \in x$. Note that $\sum x_i = S$, so decreasing x_i to lower $f_i(x_i)$ increases in the value of a different coordinate x_j and raises $f_j(x_j)$ because the

```
Randomly choose an $x \in V$ where where $\sum x_i = S$.
Sort the components of $f(x)$ into decreasing order and relabel the indices.
Let $j = P - 1$.
while $j > 0$ do
    if $f_j(x_j + 1) < f_0(x_0)$ then
        Bisect the interval ($x_j, x_0$) and increase (decrease) $x_j$ ($x_0$) by that amount.
        while $f_j(x_j) > f_0(x_0)$ do
            Set $x_0 = (x_j + x_0)/2$.
            Bisect the interval ($x_j, x_0$) and increase (decrease) $x_j$ ($x_0$) by that amount.
        end while
        Sort the components of $f(x)$ into decreasing order and relabel the indices.
        Let $j = P - 1$.
    else
        Set $j = j - 1$.
    end if
end while
```

Figure 4.2: The Generalized Binary Search Pseudocode.

function f is monotonic. We will now prove that the algorithm finishes in a finite number of steps and that the modified terminal vector is optimal over all row-based distributions. The algorithm is shown in Figure 4.2.

Theorem 3. *The (outer) while loop of the GBS algorithm must terminate.*

Proof. Trivially, if $f_j(x_j + 1) \geq f_0(x_0)$ for all $j = 1 \ldots P - 1$, the loop ends in $P - 1$ steps.

Otherwise, whenever $f_j(x_j + 1) < f_0(x_0)$, x_0 and x_j are modified such that the completion time either (1) improves or (2) does not change. In the former case, the improvement is at least equal to the minimum of $f_0(x_0) - f_j(x_j)$ and c_0. The new largest component is either the first or the j^{th} component, depending on how much the first component decreased. Either way, in at most P steps, the improvement in completion time is equal to the minimum of the slopes of the linear components. Clearly this rules out the possibility of an infinite loop, or else the completion time could be improved to zero in a finite number of steps.

In the latter case, the first and j^{th} components (and possibly other components) are equal. Hence, one possibility is that the first component will be reduced, and then the second, and so on, until eventually there will be a reduction in the overall completion time. This reduces to the above case. If there is never a reduction, the optimal value will be achieved in P steps. □

The algorithm described above is polynomial. Suppose the original maximum is M and each reduction is c, where $c = min(c_i)$. After P iterations of the outermost while loop, the improvement must be at least c. It would take time $(M/c)\dot{P}^3$ (assuming a worst case sort of P^2) to reach zero. Thus, this is an efficient algorithm.

Theorem 4. *The modified terminal vector is optimal.*

Proof. By contradiction. The output of the algorithm is a vector x such that when sorted, $f_0(x_0)$ is the maximum. Assume a vector $y \in V$ exists where $\sum y_i = S$ and the completion time is superior to that with x. When y is sorted, clearly y_0 must be one unit smaller than x_0. $\sum y_i = S$, so there must be a component of y, say y_i, that is at least one unit larger than x_i, the corresponding component of x. However, the corresponding value $f_i(y_i)$ must then exceed the value of the maximal first component of x, $f_0(x_0)$, so $f(y)$ cannot be minimal. □

4.3.2 Greedy Search

The greedy search (*GS*) algorithm we implemented relies on the monotonicity of the evaluation function f. It does not sort the elements of the vector $x \in V$, but instead finds the elements of x with the maximum and minimum f-values (i.e. x_u and x_d such that $f_u(x_u) = \max f_i(x_i)$ and $f_d(x_d) = \min f_i(x_i)$) during a single scan over all the elements. It then adjusts x_u and x_d, if possible, to reduce $f_u(x_u)$. This step will increase $f_d(x_d)$ because $\sum x_i = S$, but this still results in an overall decrease in execution time. Once done, it repeats this process.

The *GS* algorithm is fast because it quickly minimizes the maximum f-values of the elements in the vector. However, it cannot be optimal as it does not perform a pairwise comparison of every element with every other element. For example, suppose in vector x there is a third element x_a, such that its f-value is smaller than $f_u(x_u)$ but much larger than $f_d(x_d)$ (i.e. $f_u(x_u) > f_a(x_a) >> f_d(x_d)$). Further, suppose $f_u(x_u{-}1) < f_d(x_d{+}1)$, but $f_u(x_u{-}1) > f_a(x_a{+}1)$. The *GS* algorithm will not compare the data in nodes u and a, and thus will not detect that moving data from node u to a would result in a better execution time.

4.3.3 Genetic Algorithms

Genetic algorithms (*GA*s) are a heuristic search approach based on the premise that, as described in the theory of evolution, by combining parts of currently existing "good" solutions an even better one can be generated. A set of candidate solutions that exist at a certain point in time is called a "generation". The search algorithm consists of iteratively producing new generations of candidate solutions until either a limit to the number of generations is reached or the improvement of the solutions from one generation to the next is below a threshold.

The algorithm starts with an initial population of X members generated at random. Each member is evaluated using an objective function (MHETA in our case). The two members with the highest values (the "parents") are kept and the remaining $X - 2$ members are created by combining portions of the solution of the parents. New candidates are created by selecting a "crossover" point in which portions of one candidate are combined with the other. The *GA* we implemented allows for multiple (two or more) crossover points. Note as the number of crossover points increases, the variety of generated candidates can also increase. Thus, the heuristic can examine and evaluate a larger search space. As new generations are created, however, an increasing number of the population will become homogeneous, which corresponds to the algorithm "converging" onto a solution.

Recall from the previous section that the data distribution problem has an additional constraint on the possible solutions—namely, given a distribution $X = \{x_1, \ldots x_N\}$ over a dimension of size S, $\sum x_i = S$. A naive representation would place in x_i the number of rows assigned to the i^{th} processor. However, this approach easily causes problems during the crossover process. In particular, the *GA* can generate solutions where $\sum x_i \neq S$, as shown on the top of Figure 4.3.

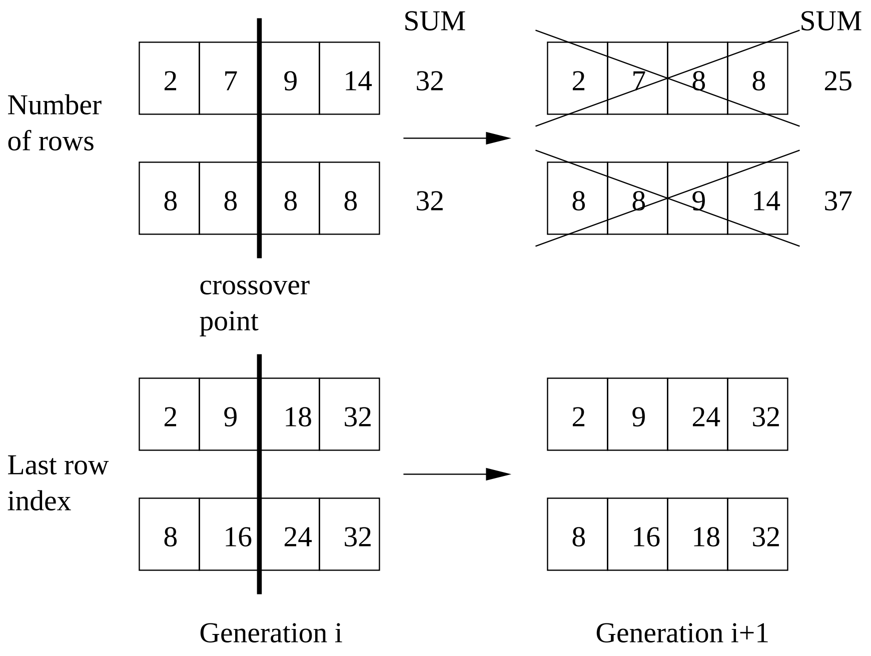


Figure 4.3: Two different representations for solutions in genetic algorithms. The top figure shows where each element in the vector stores the number of rows the nodes process, which is problematic after the crossover function is applied. The bottom figure shows a different representation, where each element stores the index of the last row the node for that index processes. This latter approach generally maintains the constraint that all elements of the vector must sum to S. Any that are not valid are not included in the future generation.

The values of the vector elements must now be resized to correct this problem, which can cause unintentional perturbations in the entire vector. We used an alternative approach in which each element stores the index of the last row assigned to a node. Shown at the bottom of Figure 4.3, this representation removes the need for resizing the elements after performing the crossover operation. However, we must now consider an ordering constraint (i.e. $x_0 \leq x_1 \ldots \leq x_N$), and we drop any generated solutions that do not satisfy this constraint. Note that with the exception of x_0 (which already contains the number of rows assigned to the 0^{th} node) we can simply determine the number of rows for the i^{th} node as $x_i - x_{i-1}$.

One of the drawbacks of this approach is the susceptibility it has to the initial conditions, such as the population size. With poorly selected initial conditions, the *GA* can converge onto a local optimum very quickly and then not improve because a majority of the population are that solution. We implemented a common solution to counter this problem in which we introduce a “mutation” factor. Given a certain probability set by the user, the *GA* randomly selects a portion of the solution that is being generated and modifies it slightly. If this mutation results in a candidate with a higher “fitness” than the majority of the other candidates in a generation, the *GA* can leave the local optimum Note that if the mutation probability is too high, however, the behavior of the *GA* will essentially become similar to that of a random search.

4.3.4 Simulated Annealing

Simulated annealing (*SA*) was first proposed by [37] as an algorithm to simulate the annealing process. A thorough examination of *SA* operations can be found in [20], and a brief summary is given here. A *SA* is a generalization of a Monte Carlo method for examining the equations of state and frozen states of n-body systems. The energy levels of the bodies shift randomly, where the probability of changing from one state to another is determined by a function of the current and new energy states (ϵ and ϵ', respectively), the temperature (T) and the Boltzmann’s constant (K_B):

$$P(\epsilon, \epsilon', T) = e^p \text{ where } p = \frac{(\epsilon - \epsilon')}{(K_B \cdot T)} \tag{4.1}$$

Intuitively, the probability of an energy level decreasing is generally very high, but there is also a non-zero chance that it may increase; this probability is greater with high temperature values. The annealing process thus starts with the system state at a high temperature, where there are many bodies will accept increased energy levels. The state is then gradually cooled, and the material becomes increasingly ordered until it solidifies. The process can be thought of as an adiabatic approach to find the lowest energy state, and therefore can be adapted to be a constraint based optimization technique.

Standard *SA* algorithms are general purpose optimization techniques described by the pseudocode in Figure 4.4. We implemented a version that is tailored to the data distribution problem

```
1:  Create initial solution $S$
2:  Initialize temperature $T$
3:  while no changes in $F(S)$ do
4:      for $nIters$ iterations do
5:          Generate a random transition from $S$ to $S'$
6:          if $F(S) \leq F(S')$ then
7:              $S \leftarrow S'$
8:          else if $e^p > random[0,1]$ where $p = \frac{(F(S)-F(S'))}{(K_B \cdot T)}$ then
9:              $S \leftarrow S'$
10:         end if
11:     end for
12:     Reduce $T$
13: end while
```

Figure 4.4: General pseudocode for the *SA* algorithm.

by modifying the transition function in Step 5 of the pseudocode. At each temperature, an *SA* randomly changes the elements of the initial vector S to produce alternative vectors S'. Our version instead produces S' using a greedy strategy similar to the *GS* algorithm. MHETA generates the F-values for each x_i element in isolation, and we find the element x_M where $f_M(x_M) = \max f_i(x_i)$. A random number of rows are then removed from x_M and proportionally reassigned to the other elements. This reassignment is such that nodes with smaller F-value receive more rows than those with higher F-values. The conditional acceptance of this alternative vector is still the same—if $F(S') < F(S)$, $S \leftarrow S'$ with 100% probability, or if $F(S') > F(S)$, S' will be accepted with the probability from Equation 4.1. If the predicted execution time does not drop by $nIters$ number of iterations, then we create another initial start vector by (1) setting $x_M \leftarrow 0$, and then (2) reassigning the x_M rows to the other vectors.

The primary disadvantages to using an *SA*, even one that is tailored to our particular problem, are its sensitivity to (1) initial conditions and (2) cooling schedule. If the initial temperature of the system is not high enough, the system will not start in a chaotic enough state. Either a low initial temperature or a cooling rate that is insufficiently slow may result in the system becoming trapped in a local minimum energy state.

CHAPTER 5

PREDICTING EXECUTION TIME USING COMPUTATIONAL MODELS

The *GBS*, *GS*, *SA*, and *GA* search algorithms described in the previous chapter require an application's execution time for each candidate data distribution. Actually running the application with this distribution is accurate, yet highly impractical due to its long execution time, and these search algorithms must evaluate *many* distributions on the fly. Therefore, we designed and implemented a computation model called MHETA (Model of HETerogeneous Architectures) to predict an application's execution time quickly and accurately. It is a system of equations that incorporates information about the application structure and the behavior and costs of running it on the underlying architecture. It can then input a data distribution and output a predicted execution time for the application.

We first present the framework of our computation model in Section 5.1. The methods used to extract application structure and the parameter values for MHETA are presented in Section 5.2 followed by development of MHETA in Section 5.3. We show MHETA's accuracy in predicting execution times in Section 6.2.

5.1 COMPUTATION MODEL FRAMEWORK

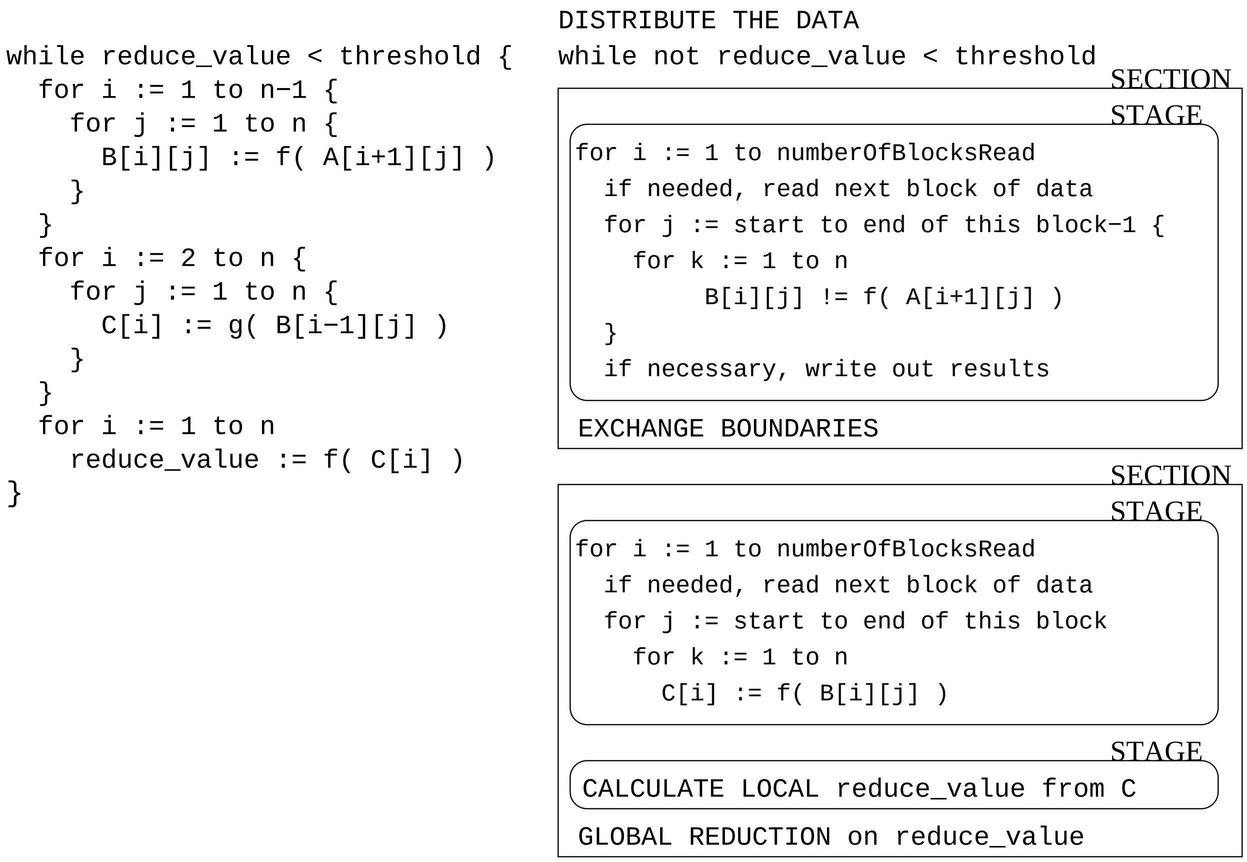
We use a computational model that is similar to BSP [55]. The programs we support are iterative scientific applications. A data distribution that reduces the execution time per iteration will consequently reduce the execution time for the entire program, so we focus on modeling the execution of a single iteration.

We define a *parallel section* as code between two *communication events*. Messages sent during a communication event are assumed to be of uniform size to simplify the model design, but MHETA can support differing sizes with more sophisticated instrumenting techniques.

A parallel section consists of a set of one or more *tiles*; pipelined applications have multiple tiles per parallel section. A tile is further divided into one or more *stages*, which are bounded explicitly by an outermost loop over a multidimensional array or implicitly by the end of a tile. MHETA can support the case where iterations take a nonuniform amount of time; however, the discussion here is only for those whose time is uniform, which covers many, if not most, applications[1]. We assume the applications make explicit calls to read and write from disk and are constructed so that data passes through memory as few times as possible. However, note that our model uses stages to handle the case where there need be multiple disk reads and writes in a parallel section. Asynchronous I/O in the form of prefetching is done by unrolling the outermost loop. Figure 5.1 illustrates an example of parallel sections and stages. The stages measure the duration to perform the computation and I/O in a program, and a parallel section sums these before including the communication cost. Any communication is assumed to involve possibly asynchronous sends but synchronous receives, which may potentially be restrictive when applied to scientific applications that do in fact use asynchronous sends and receives.

We adopt the terminology for out-of-core applications from [9]. An application is considered to be *in core* when all disk accesses for primary data sets are compulsory—this occurs when each node has its primary data set in its local memory for the duration of the program (after compulsory reads). We call the subset of the input and working data that is stored locally on a machine its Local Array (LA). An application is necessarily *out of core* if the dataset is larger than total aggregate memory. It is also out of core if a particular data distribution results in at least one node whose LA cannot fit into its memory. If either is the case, the LA is called the Out-of-Core Local Array (OCLA), and the subset that fits into memory is called the In-Core Local Array (ICLA). A node

[1]Primarily, this simplifies the implementation, as the duration to perform computation per iteration would otherwise be need to be collected for nonuniform iterations.



```
while reduce_value < threshold {
  for i := 1 to n−1 {
    for j := 1 to n {
      B[i][j] := f( A[i+1][j] )
    }
  }
  for i := 2 to n {
    for j := 1 to n {
      C[i] := g( B[i−1][j] )
    }
  }
  for i := 1 to n
    reduce_value := f( C[i] )
}
```

```
DISTRIBUTE THE DATA
while not reduce_value < threshold
for i := 1 to numberOfBlocksRead
  if needed, read next block of data
  for j := start to end of this block−1 {
    for k := 1 to n
         B[i][j] != f( A[i+1][j] )
  }
  if necessary, write out results
EXCHANGE BOUNDARIES
for i := 1 to numberOfBlocksRead
  if needed, read next block of data
  for j := start to end of this block
    for k := 1 to n
      C[i] := f( B[i][j] )
CALCULATE LOCAL reduce_value from C
GLOBAL REDUCTION on reduce_value
```

Figure 5.1: A sample conversion of a sequential program into a parallel one with sections and stages.

whose local array is too large needs to read and process the entire OCLA in ICLA-sized pieces. An in-core application incurs a single disk read for each of its local arrays, whereas an out-of-core application incurs multiple reads (and possibly writes) for each local array. Note that the problem of determining which arrays are out of core is orthogonal to our research—we currently use a simple heuristic and are primarily interested in creating a model that finds an effective partitioning of data and computation to each node.

We assume that a one-dimensional data distribution is used, and the data is divided into variable-sized blocks (called `GEN_BLOCK` in HPF [25]); each node receives its block and stores it on its local disk. We use the owner computes [27] and Local Placement [9] rules, in which nodes update data elements that reside in their local disk, but can reference other elements.

5.2 Extracting MHETA Components

MHETA requires knowledge of a program's structure and runtime-specific information—such as the costs to perform I/O, communication and computation—to calculate its execution time. Some information is extracted via static analysis and microbenchmarks, while other information requires an instrumented run of a single iteration of the application. We currently analyze the application source code manually to determine its structure, such as the number and relationship between the parallel sections, tiles, and stages as well as which variables they use. We store this information in a file read by MHETA. Section 5.2.1 gives a general formalization of the process we used into an algorithm that can be incorporated into a pre-processor to do this automatically.

Basic communication costs, such as send and receive overheads and latencies per byte to send data from one node to another, are independent of runtime conditions, so we measure them using microbenchmarks. These values are stored and referenced when needed. The instrumented run outputs computation and I/O costs and the participants in communication at the beginning or end of a parallel section. Section 5.2.2 details how we automated some of the measurement of the costs of computation and I/O. Some discussion of extracting the identities of nodes (*nIDs*) during a communication pattern follows in Section 5.2.3.

5.2.1 Extracting Application Structure

The target applications that MHETA models are iterative. After initialization, the program loops over the computation and communication multiple times, with a global reduction commonly at the end of each iteration. We currently disregard program initialization because it is a small part of the total execution time. We first describe how we determine a communication event that is used to define the beginning and end of a parallel section. This is followed by the algorithm to detect a parallel section, and a stage in isolation, then with tiles included. This section ends with some discussion of how to extract variable name information associated with a specific stage.

We assume that the application communicates via MPI. A *communication event* essentially involves invocation of one or more MPI functions. These functions range from the most basic involving a single sender and receiver (MPI_Send or MPI_Recv), to implementations of more complex but well-known communication patterns, such as MPI_Bcast for a broadcast operation. The programmer can alternatively design and implement a nonstandard pattern, such as a shuffle, typically as a loop iterating through a series of send and receive function calls. This loop could further be contained within its own function, which is called when the communication pattern defined in the function is needed. The pre-processor can therefore define the boundaries of a communication event by starting at an MPI function call and extending that definition outward until reaching one of three conditions:

- Loop bounds - the communication event encapsulates instructions from the start to the end of the loop.
- Function boundaries - the programmer created his own function for a specific communication pattern used in the program, so the event includes the entire function definition.
- Neither of the first two conditions are met and it encounters statements that are not MPI function calls - the communication event consists of a single function call.[2]

Figure 5.2 shows examples of communication events as defined above.

Given a clear definition of a communication event, we detail here how a pre-processor divides an application without pipelined communication into parallel sections and stages. The deterministic finite automaton in Figure 5.3 shows the pre-processor's states and transitions for this simple case, which does not allow for nested loops and the stage is explicitly bound by the start and end of loops. At the start of the main iterative loop, the initial state is Parallel Section with $section \leftarrow 0$. If a communication event is encountered while in this state, the current parallel section is over and

[2]This is common in nearest neighbor and pipelined communication.

MPI Functions in a loop

```
for i = 0 to num_neighbors {
  MPI_Send( <to neighbor[i]> );
}
```

User−defined Function

```
my_comm_function( <args> ) {
    for i=0 to num_neighbors/2 {
        MPI_Send( <to neigbors[i*2] );
    }
     for i=0 to num_neighbors/2 {
       MPI_Recv( <from neighbors[i*2−1] );
    }
}
```

Single MPI Function

```
<computation code>
MPI_Bcast( <arguments> );
<computation code>
```

Figure 5.2: Examples of communication events.

another starts ($section$++ and $stage \leftarrow 0$). The pre-processor enters and exits the stage state when it encounters the start and end of a loop in the soure code, respectively. The pre-processor then returns to the Parallel Section state, when it can enter another stage and the $stage$ variable is incremented. If a loop ending boundary is reached while in the Parallel Section state, the iterative loop is over, and the pre-processor is done. The state diagram is considerably more complex for when we allow nested loops inside stages and want to define tiles. These complications are detailed in Appendix A, where we use the more powerful nondeterministic finite automaton state machine to guide the pre-processor.

The pre-processor can further analyze the code of a stage to determine which variable is to be used as the representative of the costs. There are two complications in this analysis:

- There are variables that are distributed and those that are *replicated*. Stages with replicated variables do not change their computation or I/O costs with differing data distributions, whereas these costs can be drastically different for distributed variables.

- Some stages may not contain any I/O function calls or any loops at all, in which case a suitable representative may not exist; which variable is selected is not important in this case because the computation cost is constant.

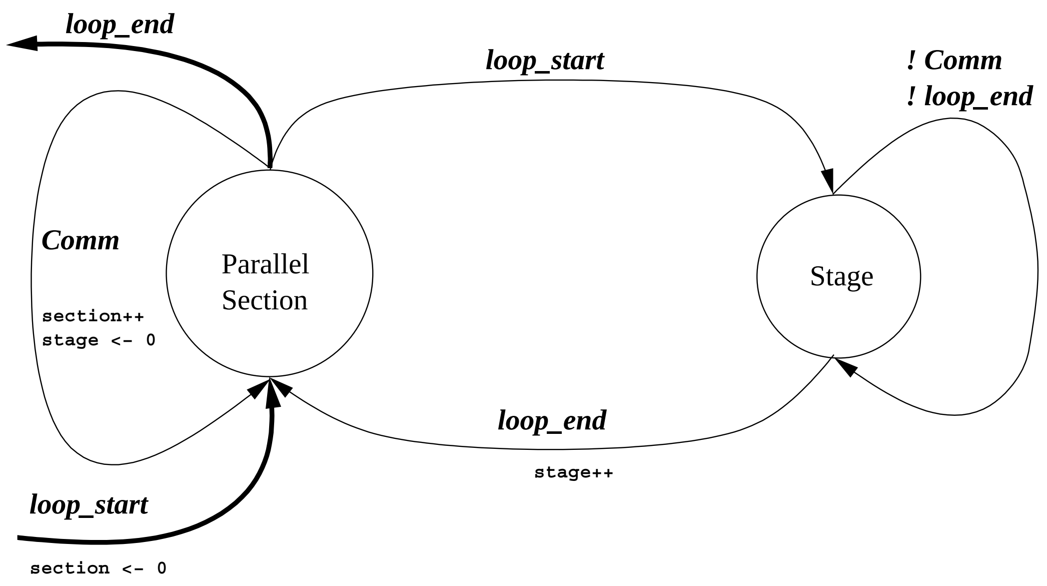


Figure 5.3: A deterministic finite automaton showing the states and transitions for the instrumenting pre-processor when analyzing an application to break into parallel sections and stages. The ***italic and bold*** terms are statements (tokens) encountered in the source code. `Courier and bold` terms are actions for internal variables.

In stages that have both distributed and replicated variables, preference should be given to the distributed arrays because these are likely to be out of core and dominate computation and I/O costs.

5.2.2 Measuring Runtime Costs

This section starts with details of how we measure synchronous and then asynchronous I/O costs. The difficulty in measuring computation durations is discussed next, as this measurement requires knowledge of when a stage begins and ends, and so is hard to detect in general. MHETA targets applications that use explicit I/O, so these programs have explicit function calls to read and write data. For programs that use MPI function calls to perform I/O, we use the MPI Jack tool [1], an interface that exploits PMPI, the profiling layer of MPI. MPI Jack enables a user transparently to intercept any MPI call and execute arbitrary code before and/or after an intercepted call. These are

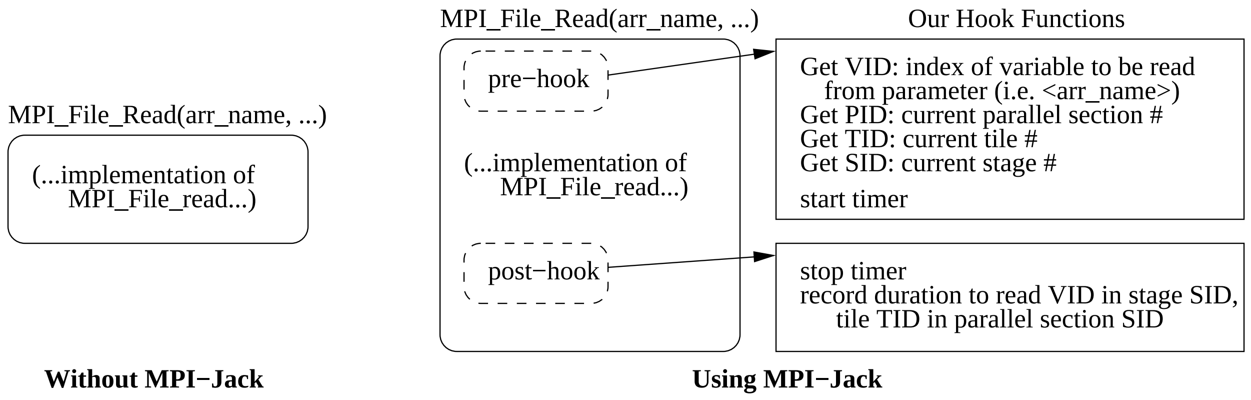

MPI_File_Read(arr_name, ...)
(...implementation of MPI_File_read...)
Without MPI−Jack
MPI_File_Read(arr_name, ...)
pre−hook
(...implementation of MPI_File_read...)
post−hook
Using MPI−Jack
Our Hook Functions
Get VID: index of variable to be read from parameter (i.e. <arr_name>)
Get PID: current parallel section #
Get TID: current tile #
Get SID: current stage #
start timer
stop timer
record duration to read VID in stage SID, tile TID in parallel section SID

Figure 5.4: A graphical example of how MPI Jack is used to extract timing information for an MPI_File_read call. Invoking a MPI Jack function essentially calls the pre and post hook functions before calling the actual MPI function. On the left, these hook functions are undefined and no timing information is recorded. The right implementation has code defined to extract the variable ID involved in the I/O and timers to record the latency to read the array.

called *pre* and *post* hooks. An example of an MPI read operation is shown in Figure 5.4. The seek overheads for reading and writing (O_r and O_w) are the same regardless of the variable involved, so they are measured and output as node-specific data. The corresponding latencies ($T_r(m)$ and $T_w(m)$), in contrast, are specific to the variable m. The instrumenting code extracts the variable ID from the function's parameters to record the latencies with the corresponding variable. The runtime system computes the latencies ($r(m)$ and $w(m)$) for a single element of m and stores them and overhead costs into a MHETA internal file.

Note that given a data distribution for the instrumented execution, the size of m may in fact be in core, and no I/O occurs. MHETA would thus effectively have a zero latency associated with reading or writing this variable. This of course would be incorrect for distributions that assign more of m to the node than it can store in memory, thereby causing I/O to occur. We ensure that the correct latency values are measured during the instrumented run by forcing *all* nodes to perform I/O for any potentially out-of-core variables.

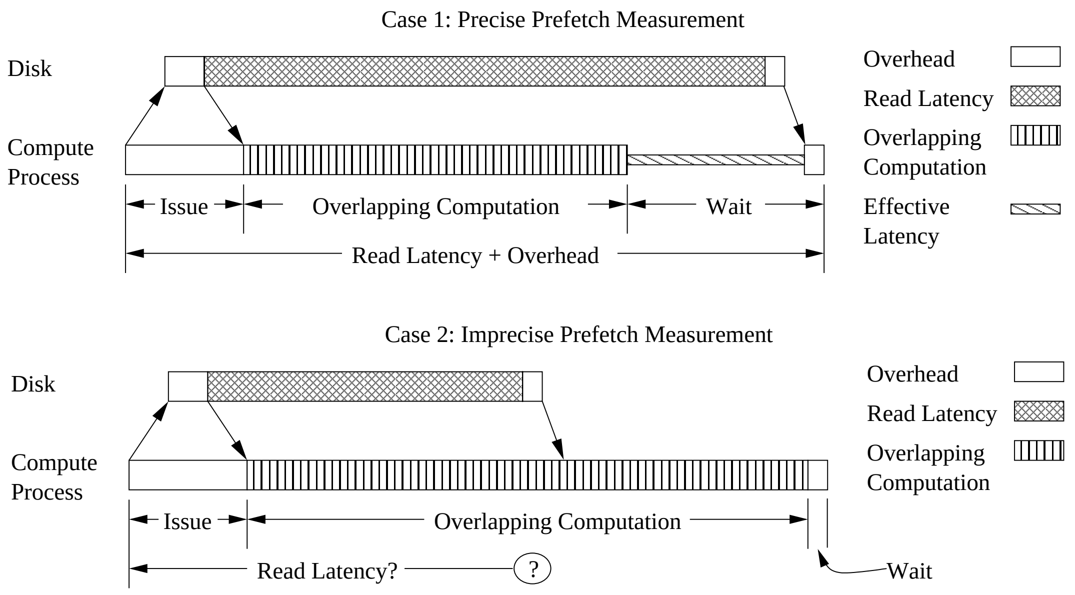


Figure 5.5: Problems when trying to instrument prefetching code. The top figure shows how instrumenting prefetching can be precise and accessible from our wrapper functions. The overlapping computation is timed from the end of the prefetch issue to the beginning of the wait function, and the disk read latency is timed from the beginning of the issue to the end of the wait function. Unfortunately, as the bottom figure demonstrates, if $T_A > T_r(m)$, we cannot compute $T_r(m)$ correctly via timers in the wrapper functions.

Unlike synchronous I/O, asynchronous I/O allows overlap of computation and I/O costs, so predicting the cost of prefetching requires a combination of $T_r(m)$ and the duration of the overlapping computation (T_A). Figure 5.5 shows that T_A is the time from the prefetch issue until the entry to the wait function, and this duration can be measured using timers. The difficulty arises when we try to measure the duration of $T_r(m)$. If $T_A \leq T_r(m)$, $T_r(m)$ is measured from the start of the prefetch issue to when the wait function returns. Otherwise, there are no timers the runtime system can use to determine when the I/O operations finish.

Our approach to solve this problem cleanly divides the operations by forcing (1) all prefetch issues to be the same as a *synchronous read* and (2) wait functions to be no-ops, as shown in Figure 5.6. This technique is simple to implement and accurately measures the durations of T_A and

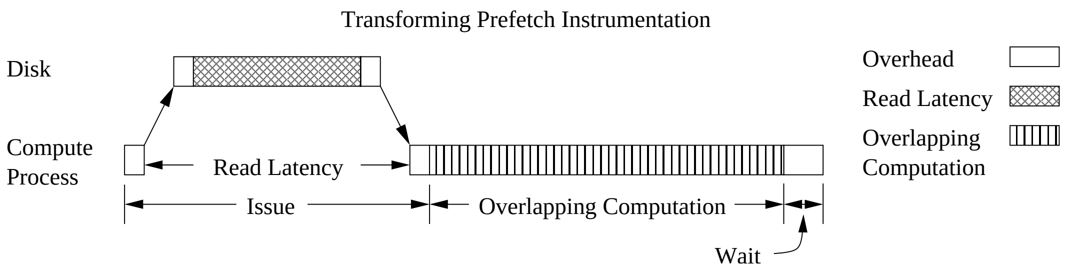


Figure 5.6: Instrumenting prefetching by forcing issues to be blocking reads, and transforming "wait" functions into no-ops.

$T_r(m)$. We assume only one iteration is instrumented (out of many), so the higher latencies experienced using this technique are amortized over the remaining iterations that do standard prefetching.

Measuring the computation time for a stage (a separate duration from T_A) is more complex to automate than I/O; the computation boundaries are difficult to detect in the source code. However, it is possible with help from a pre-processor that determines stage boundaries, though we have not implemented such a pre-processor at this time. Only computation and I/O occur within a stage; by measuring the duration of the stage, the computation duration can be calculated as the total time for the stage minus time to perform I/O.

5.2.3 Extracting Communication Participants

Communication events involve sending and receiving messages to and from various nodes. MHETA requires information about the sender and recipient *nIDs*. The node IDs are available from the parameters of the MPI send and receive calls. Therefore, via the pre hooks in MPI Jack, we can get this information, making this transparent to the programmer.

5.3 MHETA Implementation

We implemented MHETA as a series of equations for each parallel section and stage. Given values for computation, I/O and communication costs specific to an application running on a particular architecture, MHETA calculates the total execution time of a single iteration of the application given an *arbitrary* data distribution.

We begin by considering the execution of a single parallel section on one node, first with a single stage and then multiple stages. We develop the costs of first synchronous and then asynchronous I/O in a particular stage. Next, we consider the communication involved when the program is first distributed between two nodes, followed by multiple nodes. Finally, we conclude with the overall MHETA model—multiple parallel sections in a multi-node program.

5.3.1 Single Node and Parallel Section

First, we describe how MHETA will generate the predicted time to perform only computation and I/O in a single stage in a single parallel section. We already know the duration to perform computation on the data given by a distribution δ during the instrumented run of the program. For each distribution in the set of candidate distributions, $\delta' \in \mathcal{D}$, predicting the new computation time is relatively straightforward: it is the original time, T_ϕ, multiplied by the ratio of the new amount of work W' and the work assigned to it by δ, which is W:

$$(T_\phi)' = T_\phi * \frac{W'}{W}$$

Modeling the time taken to perform I/O is more complicated because we need to accurately calculate the cost of disk accesses due to out-of-core arrays. MHETA currently uses a simple heuristic to determine if an array is out of core for δ'. This heuristic is based on the intuition that arrays with smaller sizes or those that are accessed more frequently are preferred to be kept in core. Once a variable m is determined to be out of core, MHETA calculates its ICLA ($M_\iota(m)$) and OCLA sizes ($M_\beta(m)$). We can then compute the number of times the disk is accessed in this stage for an out-of-core variable m by taking the ceiling of dividing its OCLA size by its ICLA size:

$$N_L(m) = \left\lceil \frac{M_\beta(m)}{M_\iota(m)} \right\rceil$$

Any time the node reads data from disk, there is a corresponding write to disk if the results of the computation are stored, such as in our Jacobi application. For applications such as Conjugant Gradient and Lanczos, the array is read-only, and no writes are performed.

The total time spent performing *synchronous* I/O for an out-of-core array m in this stage, $T_\psi(m)$, is the time to perform the reads and writes of its ICLAs multiplied by the number of times it is performed. These times include the overheads to prepare the reads (O_r) and writes (O_w) and the respective latencies ($T_r(m)$ and $T_w(m)$):

$$T_\psi(m) = N_L(m) \cdot [(O_r + T_r(m)) + (O_w + T_w(m))] \tag{5.1}$$

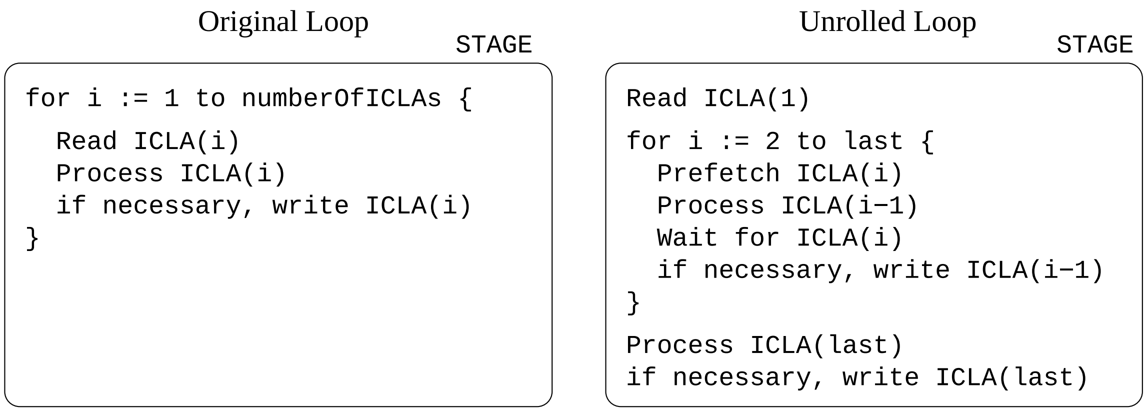
where $T_r(m) = r(m) \cdot M_\iota(m)$ and $T_w(m) = w(m) \cdot M_\iota(m)$. Note that if variable m is in core, then $T_\psi(m) = 0$, and if the array is read-only, $O_w = T_w(m) = 0$.

Equation 5.1 assumes that the full disk latency is incurred on every read; i.e. $T_r(m)$ is the time to read $M_\iota(m)$. Programmers and/or compilers, however, can insert asynchronous read requests (prefetching) to hide some of this latency. The *effective* read latency, $T_s(m)$, is calculated in MHETA as $T_r(m)$ minus T_A, the time the node spends computing between the prefetch issue and when the data is needed. If the overlap computation is greater than or equal to the read latency, $T_A \geq T_r(m)$, this latency has been effectively masked: $T_s(m) = 0$. We assume that the program does not perform asynchronous writes because the latency can be masked by the operating system using write-back buffers. Equation 5.1 becomes:

$$T_\psi(m) = N_L(m) \cdot [(O_r + O_A + T_s(m)) + (O_w + T_w(m))] \tag{5.2}$$

where $T_s(m) = \max[0,\ T_r(m) - T_A]$. One problem in using the asynchronous I/O facilities is that extra overhead is incurred (O_A) regardless of whether the prefetch was issued early enough to hide the read latency. Prefetching can thus be more expensive than regular (synchronous) reads. However, note that when the application does not use prefetching, Formula 5.2 reduces to Formula 5.1 because $T_s(m) = T_r(m)$ and $O_A = 0$.

The above formula can be modified to take into account how typical pipelined asynchronous programs can be rewritten (e.g. Morly [39]). A loop (see Figure 5.7) over ICLAs that involves



```
for i := 1 to numberOfICLAs {
  Read ICLA(i)
  Process ICLA(i)
  if necessary, write ICLA(i)
}
```

```
Read ICLA(1)
for i := 2 to last {
  Prefetch ICLA(i)
  Process ICLA(i−1)
  Wait for ICLA(i)
  if necessary, write ICLA(i−1)
}
Process ICLA(last)
if necessary, write ICLA(last)
```

Figure 5.7: A sample transformation of a loop over ICLAs to enable prefetching.

a read, followed by computation on that ICLA, and finally a write-back of the results to disk is slightly unrolled. The stage boundary of course gets extended as necessary. The program can now prefetch the ICLA for the $i+1^{th}$ iteration while performing computation on the ICLA for the i^{th} iteration. The first ICLA read incurs the full latency, whereas the remaining $N_L(m)-1$ read latencies can be mitigated using prefetching. All overhead costs are incurred regardless of the loop structure. The adjusted formula taking into account this type of prefetching program structures is:

$$T_\psi(m) = [N_L(m) \cdot (O_r + O_A + O_w + T_w(m))] + T_r(m) + [(N_L(m) - 1) \cdot T_s(m)]$$

The total amount of time spent in computation and I/O for this stage is therefore the time for computation (T_ϕ) plus the sum of all accesses to disk. $T_\psi(m)$ is non-zero only when m is in β, the set of out-of-core variables for the current data distribution.

$$T_\gamma = T_\phi + \left[\sum_{m\in\beta} T_\psi(m)\right]$$

Once the duration for the k^{th} stage can be calculated and assuming a parallel section contains N_γ stages, the time a node spends in the parallel section (T_τ) performing computation and I/O is:

$$T_\tau = \sum_{k=1}^{N_\gamma} T_\gamma(k)$$

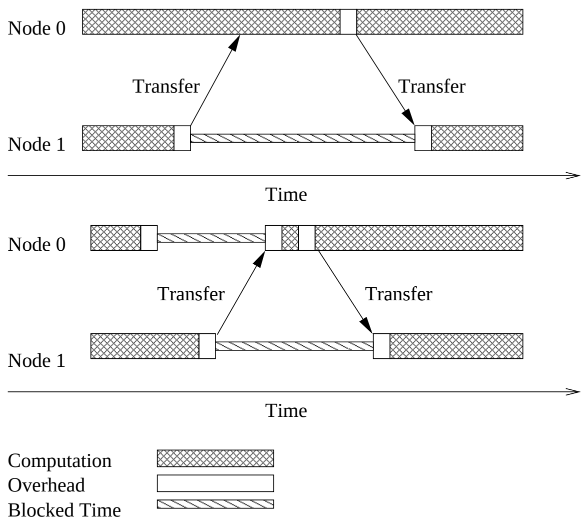


Figure 5.8: A graphical description of a reduction. The top figure shows when node 1 is blocked waiting for node 0 to send its message, and the bottom shows the situation when node 0 waits for node 1's message, and node 1 waits for node 0's response.

5.3.2 Extension to Two Nodes

When considering multiple nodes, MHETA needs to include communication costs. Assume two nodes, P_0 and P_1. The communication will be first broken into the components involved for point to point communication, and they are then combined together in MHETA to calculate the time taken to perform communication according to a specific pattern. Given a message q, the communication cost is broken into three components:

- $O_{\rightarrow}(q)$ - the overhead incurred when sending,
- $T_w(i, q)$ - the time P_i spends when potentially waiting for a message, and
- $O_{\leftarrow}(q)$ - the overhead when a node processes an incoming message.

An example of the relationship between these parts is shown in Figure 5.8.

The value $O_{\rightarrow}(q)$ is a combination of two costs. There is always the fixed overhead ($O_S(q)$) to prepare and actually copy the message into a system buffer. Furthermore, for arrays from which the message is generated is out of core, the node needs to read it from disk (i.e. $q \in \beta$, the set of out-of-core variables); the same modeling is done here as described for I/O above. Therefore, $O_{\rightarrow}(q)$ is the sum of the fixed overhead plus the time to read the message of size M_q from disk:

$$O_{\rightarrow}(q) = O_S(q) + (O_r + r(M_q) \text{ if } q \in \beta)$$

We next compute the value of $T_w(i,q)$. Nodes participating in non-pipelined applications generally start to block *after* performing their stages. Thus, the time P_1 spends waiting for message q to arrive from P_0 is non-zero if it finishes its stages before q arrives:

$$T_w(1,q) = \max \begin{Bmatrix} T_\tau(0) + O_{\rightarrow}(q) + T_{0\rightarrow 1}(q) - [T_\tau(1) + O_{\rightarrow}(q)] \\ 0 \end{Bmatrix} \tag{5.3}$$

Note that Equation 5.3 is symmetric between P_0 and P_1 in this case, and we must now distinguish T_τ between nodes because a node's blocked time depends on the execution of the sending node. In addition to costs for send overheads and time spent waiting for a message, both nodes spend time $O_{\leftarrow}(q)$ processing the incoming message.

Pipelined applications have the property that there are potentially many tiles in a parallel section (i.e. the number of tiles, $N_T^i \geq 1$), and nodes start to wait *before* they execute the stages of each tile. Figure 5.9 shows this dependency. We now expand $T_w(0,q)$ to distinguish between tiles. Assuming that the pipeline starts from P_0, P_0 does not block at all:

$$T_w(0,q,j) = 0 : j = 0 \ldots N_T^0 - 1 \tag{5.4}$$

However, P_1 blocks at the start of its j^{th} tile if q arrives any time after it has finished all $j-1$ previous tiles. The message q is on route from P_0 after it completes all j tiles, each of which takes time to process the stages plus the send overhead ($T_\tau(0,j) + O_{\rightarrow}(q)$). For any given tile, P_1 first blocks, incurs receive overhead and then spends $T_\tau(1,j)$ time in computation. Thus the wait time for P_1 at the beginning of its j^{th} tile is the total time before q is on route, plus transfer time over the network, minus the total time for P_1 to have finished $j-1$ tiles.

$$T_w(1,q,j) = \left[\sum_{k=0}^{j} T_\tau(0,k) + O_{\rightarrow}(q)\right] + T_{0\rightarrow 1}(q) - \left[\sum_{k=0}^{j-1} T_w(1,q,k) + O_{\leftarrow}(q) + T_\tau(1,k)\right] \tag{5.5}$$

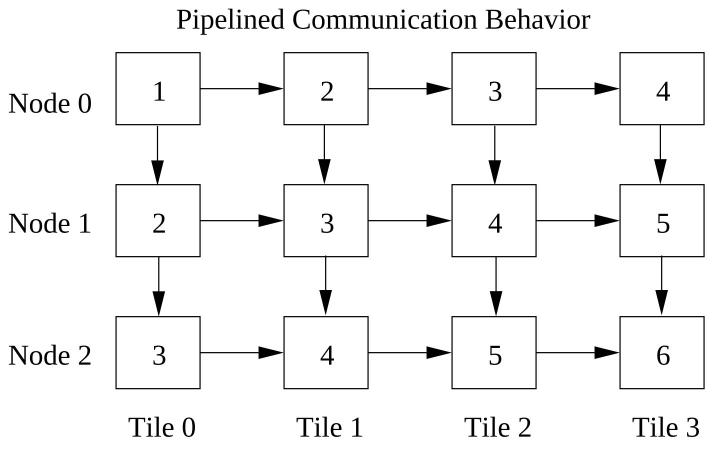


Figure 5.9: The behavior of pipelined communication. The arrows represent the dependencies of each tile on each node, and the number in each tile are the order at which the tile is processed. Note that the tiles of node P_0 have horizontal dependencies; P_0 can process them one right after the other. The tiles for P_1 have both vertical and horizontal dependencies; they depend on messages from P_0 and the tiles before it on the same node.

Of course if $T_w(1, q, j) < 0$, P_1 does not block and $T_w(1, q, j) = 0$.

We next use the specific communication pattern to calculate the duration the nodes spend in communication (denoted $T_\Upsilon(i)$) as follows:

- Nearest Neighbor: If both nodes perform their sends before blocking, the total time in communication is the sum of the send and receive overheads and the time spent waiting for the message:

$$T_\Upsilon(i) = O_{\rightarrow}(q) + T_w(i, q) + O_{\leftarrow}(q) \tag{5.6}$$

- Reduction: Given that P_0 is the master node that performs the reduction, it blocks until it has received and processed the value to use in the reduction from P_1. It then computes the value (in time T_r, which is often negligible) and prepares and sends it to P_1, which ends its time spent in communication:

$$T_\Upsilon(0) = T_w(0, q) + O_{\leftarrow}(q) + T_r + O_{\rightarrow}(q) \tag{5.7}$$

The time at which the message reaches P_1 takes into account when P_0 finishes its parallel section and sends q, and the time taken to transmit q:

$$T_\alpha(1) = T_\tau(0) + T_\Upsilon(0) + T_{0\rightarrow 1}(q) \tag{5.8}$$

The other node, P_1, sends its value used in the reduction to P_0 and then waits for the result of the reduction to return. This blocked time is the difference between $T_\alpha(1)$ and when P_1 finished its parallel section and composed and sent q: $T_\tau(1) + O_\rightarrow(q)$. Once received, P_1 then incurs overhead processing the result. The total communication time for P_1 is:

$$T_\Upsilon(1) \quad = \quad O_\rightarrow(q) + [T_\alpha(1) - (T_\tau(1) + O_\rightarrow(q))] + O_\leftarrow(q) \tag{5.9}$$

- Pipeline: P_0 has no receives and P_1 has no sends, so using the cost $T_w(1, q, j)$ from Equation 5.5 and $T_w(0, q, j)$ from Equation 5.4, the time spent in communication is:

$$T_\Upsilon(i) = \sum_{j=0}^{N_T^i} \left[T_w(i, q, j) + \begin{cases} O_\leftarrow(q) & \textbf{if } i \neq 0 \\ O_\rightarrow(q) & \textbf{else} \end{cases} \right] \tag{5.10}$$

The total execution time for node P_i in a parallel section, (denoted $T_\sigma(i)$), is the time spend performing all computation, I/O and communication:

$$T_\sigma(i) = T_\tau(i) + T_\Upsilon(i)$$

5.3.3 Extension to Multiple Nodes

Measuring communication for multiple nodes differs depending on whether the pattern is non-pipelined or pipelined. The latter kind of pattern has communication between at most two nodes, but non-pipelined communication can involve more than two. For the sake of consistency, the more complex non-pipelined communication is described first, followed by the simpler pipelined communication.

When communication costs are expanded in non-pipelined applications to include P_i sending to and receiving from multiple nodes, we introduce four more variables. Let $N_{\rightarrow}(i,*)$ and $N_{\leftarrow}(i,*)$ be the total number of messages P_i generates and receives, and $C_{\rightarrow}(i)$ and $C_{\leftarrow}(i)$ be sets of recipients and senders of its messages, respectively. In other words, $N_{\rightarrow}(i,*)$ is the number of recipients in $C_{\rightarrow}(i)$, and $N_{\rightarrow}(i,i')$ is the number of nodes that P_i has sent messages before and including a *specific* node $P_{i'}$, The values on these variables are determined at run time in our emulator and input to MHETA. Equations 5.3, 5.6, 5.7 and 5.9 for non-pipelined applications are generalized to include multiple senders and receivers, which are specified in the sets $C_{\rightarrow}(i)$ and $C_{\leftarrow}(i)$. We simplify the presentation of the more general equations by defining $T_{\rightarrow}(i,i')$ to be the time at which P_i has sent a message to $P_{i'}$ and is computed as:

$$T_{\rightarrow}(i,i') = T_{\tau}(i) + N_{\rightarrow}(i,i') \cdot O_{\rightarrow}(q) \tag{5.11}$$

The overhead to send messages is assumed to be the same regardless of who the recipient is for this communication event. The node $P_{i'} \ : i' \in C_{\rightarrow}(i)$ that is the slowest to send a message to P_i is the only one that impacts the wait time, so Equation 5.3 is modified to:

$$T_w(i,q) = \begin{cases} \max_{i' \in C_{\leftarrow}(i)} \left[T_{\rightarrow}(i',i) + T_{i' \rightarrow i}(q) - T_{\rightarrow}(i,i')\right] & \textbf{if difference} > \mathbf{0} \\ 0 & \textbf{otherwise} \end{cases} \tag{5.12}$$

Equation 5.6, which computes T^i_{Υ} for nearest neighbor communication, is extended to take into account all senders and recipients for P_i. Note the value of $T_w(i,q)$ is now from Equation 5.12:

$$T^i_{\Upsilon} = [N_{\rightarrow}(i,*) \cdot O_{\rightarrow}(q)] + T_w(i,q) + [N_{\leftarrow}(i,*) \cdot O_{\leftarrow}(q)]$$

The reduction communication pattern costs for P_0 and $P_i : i \neq 0$ are extended from Equations 5.7 and 5.9. P_0 will block until it processes all the messages from the other nodes. After computing the reduction value, it must send that value to all the nodes participating in the computation:

$$T^0_{\Upsilon} = T_w(0,q) + [N_{\leftarrow}(0,*) \cdot O_{\leftarrow}(q)] + T_r + [N_{\rightarrow}(0,*) \cdot O_{\rightarrow}(q)]$$

P_i will block until P_0 computes the reduction value and sends the value to all nodes up to and including P_i. Equation 5.8, which computes the time at which P_i receives a message from P_0, is extended to allow recipient nodes before P_i:

$$T_\alpha(i) = T_\tau(0) + T_w(0,q) + [N_\leftarrow(0,*) \cdot O_\leftarrow(q)] + T_r + [N_\rightarrow(0,i) \cdot O_\rightarrow(q)] + T_{0\rightarrow i}(q)$$

After q is transmitted over the network, P_i must then process it:

$$T_\Upsilon^i = O_\rightarrow(q) + [T_\alpha(i) - (T_\tau(i) + O_\rightarrow(q))] + O_\leftarrow(q)$$

Generalized Pipelined Communication

The node P_i will only block waiting for a message from P_{i-1} when performing pipelining. The only change from Equation 5.5 is that at each tile, node P_{i-1} can possibly block waiting for a message from all nodes before it. Let $T_p(i-1,j)$ be the time that node P_{i-1} has finished its j^{th} tile and $T_p(i,j-1)$ be the time node P_i finished its $j-1^{st}$ tile. For each tile, a node must first wait for the message from its upper neighbor, process the message, perform the computation on the tile, and finally send its message to its lower neighbor. The time when a node is finished with a tile is when it performed this sequence for all previous tiles. Therefore, we have the following values for $T_p(i-1,j)$ and $T_p(i,j-1)$:

$$\begin{aligned} T_p(i-1,j) &= \left[\sum_{k=0}^{j} T_w(i-1,q,k) + O_\leftarrow(q) + T_\tau(i-1,k) + O_\rightarrow(q)\right] \\ T_p(i-1,j) &= \left[\sum_{k=0}^{j-1} T_w(i,q,k) + O_\leftarrow(q) + T_\tau(i,k) + O_\rightarrow(q)\right] \end{aligned}$$

Again, $T_w(i,q,j)$ is non-zero only if P_i finishes its $j-1^{st}$ tile before the message from P_{i-1} arrives:

$$T_w(i,q,j) = \max\{0, T_p(i-1,j) + T_{(i-1)\rightarrow i}(q) - T_p(i,j-1)\}$$

Figure 5.10 shows this relationship between two arbitrary nodes. Note that the recurrence ends with $T_w(0,q,j) = 0$ for all j.

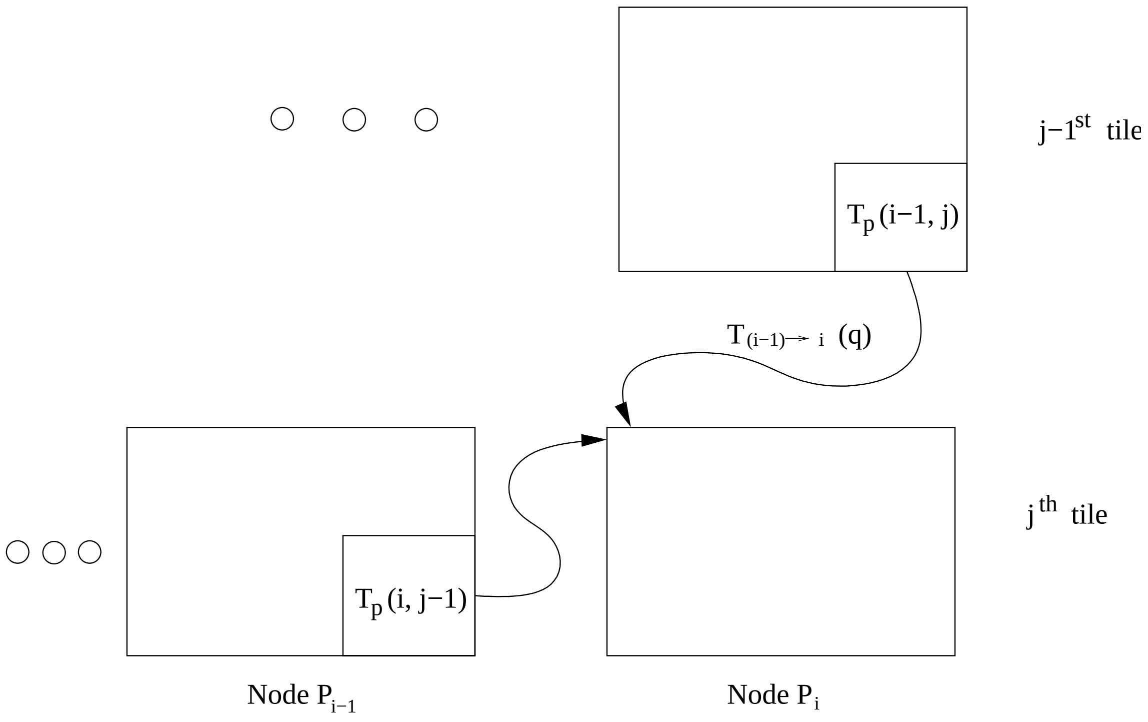


Figure 5.10: Blocked time for an arbitrary node P_i to start its j^{th} tile. The message from P_{i-1} arrives at time $T_p(i-1,j) + T_{(i-1)\rightarrow i}(q)$, and the node has finished its $j-1^{st}$ tile at time $T_p(i,j-1)$.

5.3.4 Overall MHETA Model

The time at which each parallel section is ready to send its messages in non-pipelined applications depends on how long all previous parallel sections took, so Equation 5.11 is modified to include these durations. To do this, three variables must be extended to specify that it is the j^{th} parallel section on node P_i under consideration: the variable that stores a parallel section duration, $T_\sigma(i,j)$, the variable that stores the duration for computation, $T_\tau(i,j)$, and the send set $C_\rightarrow(i,j)$. The time for the k^{th} parallel section to send its messages is thus modified to:

$$T_\rightarrow(i,i',k) = \left[\sum_{j=1}^{k-1} T_\sigma(i,j)\right] + T_\tau(i,k) + \sum_{j=0}^{i'} O_\rightarrow(q) : P_j \in C_\rightarrow(i,k)$$

The total execution time for a single iteration on node P_i ($T_I(i)$) is the sum of all its $N_\sigma(i)$ parallel sections:

$$T_I(i) = \sum_{j=1}^{N_\sigma(i)} (T_\sigma(j))$$

5.3.5 Time Complexity

The complexity of MHETA is based on the time to calculate the communication, computation, and I/O costs. For a single node P_i, the calculation of communication is bound by the number of parallel sections in the program (N_σ), the number of tiles in each of j^{th} parallel section ($N_T^{i,j}$) multiplied by the number of senders and receivers in each; this is at most the number of nodes (n). Calculating computation time is bound by the total number of stages, whereas I/O costs are bounded by both N_γ and the number of out-of-core arrays in each stage, which is at most the number of arrays in the program (N_μ). The complexity is further increased by n because execution times must be computed for all nodes. Thus, the total time complexity of MHETA is:

$$O\left(n \cdot \left(\left[\sum_{k=0}^{N_\sigma} N_T^{i,k}\right] \cdot n + \left[N_\gamma \cdot N_\mu\right]\right)\right)$$

The values of N_σ, N_T, N_γ and N_μ for a particular program will be constant, thus the time required by MHETA to compute its prediction grows quadratically with the number of nodes in the system.

CHAPTER 6

PERFORMANCE

This chapter discusses the performance of the two main components of the SHARED system that we have implemented: the data distribution search heuristics proposed in Chapter 4 and the MHETA model presented in Chapter 5. Section 6.1 describes the common setup for the experiments, such as details of the underlying emulated architectures and the types of applications we used. The effectiveness of the data distributions output by our system depends a great deal on MHETA's accuracy, and we detail in Section 6.2 our model's performance. This will follow by a discussion of the performance of each of the search algorithms in Section 6.3. General conclusions about the performance of both components will be discussed in Section 6.4.

6.1 GENERAL EXPERIMENTAL SETUP

Our underlying architecture is a homogeneous cluster of eight Dell Quad servers running Solaris 2.8. We focus on *distributed memory* architectures versus shared memory or a hybrid, so we only make use of one processor per node. We emulate (1) a slower CPU by forcing the process to do extra work, (2) a smaller main memory by forcing I/O earlier than would be necessary given the actual main memory size, and (3) slower I/O by increasing the amount of data accessed by the I/O routine. This configurable architecture is of a heterogeneous cluster as detailed in Chapter 4—the nodes can have differing memory sizes, relative CPU powers and I/O latencies.

Two scientific benchmarks are common to both sets of experiments. The first, Jacobi iteration (Jacobi), iteratively applies a five-point stencil operation on a dense matrix A and writes the results to an output matrix B until a satisfying criterion is reached. The matrix pointers are swapped after each iteration, so $B = f(A)$, for the i^{th} iteration and $A = f(B)$ for the $i + 1^{st}$ iteration, where f is the same stencil operation. The second, Conjugant Gradient (CG), was ported from the Fortran implementation in the NAS benchmark suite, and it takes a sparse matrix A and iteratively produces an output vector V for a set number of iterations. The new vector V' is generated from a combination of A and V: $V' = f(A, V)$. We also implemented and used one full-scale application, a Lanczos iterative method for solving a linear system $Ax = b$, where A is a symmetric, positive definite, $N \times N$ dense matrix, and x and b are column vectors. We selected these applications because of their iterative nature. These programs were compiled with `-O2`, and for all message passing we used LAM-MPI [41]. Jacobi has two parallel sections with one stage each and computes over two potentially out-of-core arrays (much larger than the message size), so it has a high computation to communication ratio. In contrast, CG has seven parallel sections and only two involve computation over a large array; the remaining parallel sections have computation over smaller vectors that are the same size as its messages. Thus, CG has a lower computation to communication ratio, but a higher computation to I/O ratio than Jacobi because Jacobi reads *and* writes the large arrays each iteration, whereas the array in CG is read-only. Lanczos has fifteen parallel sections, and fourteen involve computation over a potentially out-of-core array. These same parallel sections have multiple stages, some of which involve many smaller in-core vectors. The large amount of work performed between communication points results in a high computation to communication ratio for Lanczos, and the fact that the large array is read-only results in a high computation to I/O ratio. When testing MHETA's accuracy, we (1) designed and implementing another application that uses both pipelined and reduction communication patterns modeled on RNA pseudoknots (RNA), and (2) enhanced the I/O capabilities of Jacobi to include prefetching mechanisms.

We currently assume that a program's structural information (the number of parallel sections and stages as well as variable usage information) is available and input to MHETA. We instrumented our applications partially by hand and partially by using the MPI Jack interface. We performed one instrumented iteration of each application, using a `BLOCK` distribution (equal size strips assigned to each node) to measure computation time, communication overhead, and I/O cost. During the search, MHETA combines these costs with a candidate data distribution, evaluating its effectiveness. In our experiments we examine iterative scientific programs, so any absolute improvement from an effective distribution will be multiplied by the number of remaining iterations.

6.2 Measuring MHETA Accuracy

We performed detailed analysis of MHETA's accuracy on the emulated heterogeneous architecture and applications described Section 6.1. We also tested MHETA with a pipelining application (RNA) similar to RNA pseudoknots, and on Jacobi with prefetching. MHETA is on average more than 97% accurate in predicting execution times for all four programs and at least 98% accurate when taking into account prefetching in Jacobi. Our measurements show that MHETA evaluates a single distribution in about 5.4 ms. This efficiency is important because our SHARED system can potentially search through many distributions and must make a decision on the fly, and we must to keep the overhead as small as possible.

Section 6.2.1 describes the setup specific to these set of experiments, followed by comparison analysis of MHETA's prediction and actual run times in Section 6.2.2. Section 6.2.3 details some of the limitations inherent to MHETA. We conclude MHETA's analysis in this set of experiments with some discussion of the variability of execution times for an application given different data distributions in Section 6.2.4, to motivate testing our search algorithms in Section 6.3.

6.2.1 Experimental Setup for Testing MHETA

Figure 6.1 shows the distributions we tested, which ranged in two dimensions: (1) how well the load is balanced and (2) to what degree I/O costs are considered. The simplest distribution, "Block" (`Blk`), allocates data evenly across nodes without regard for I/O cost or load balance. A "Balanced" distribution (`Bal`) focuses on balancing the load on the nodes and ignores I/O costs. The "In-Core" (`I-C`) distribution in contrast ignores load and focuses on only minimizing I/O costs. The "In-Core

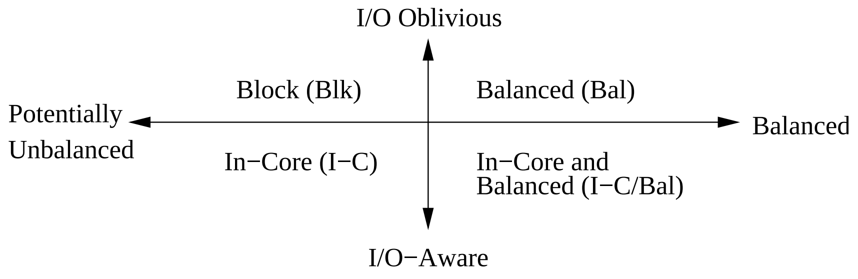


Figure 6.1: The spectrum of data distributions that we tested.

and Balanced" (`I-C/Bal`) distribution first maximizes the number of nodes that have exclusively in-core datasets and then balances the load as much as possible. We start testing the performance of MHETA with `Blk` and progressively generate distributions that move through `I-C`, `I-C/Bal`, `Bal`, and back to `Blk`. For architectures when the relative CPU power of the nodes is identical, a `Blk` distribution already balances the load, so we only vary the distribution between `Blk` and `I-C`. A similar simplification is done when the computing environment has no nodes with memory restrictions (so I/O is not a concern), where we vary the distribution only from `Blk` to `Bal`. The MHETA model extends to two-dimensional data distributions, but such distributions are problematic for run-time data distribution systems because the search space increases greatly. Hence, we focus on only one-dimensional distributions.

We emulated seventeen architecture configurations when testing MHETA's performance for all four applications, with Jacobi not performing any prefetching. For the experiments of Jacobi with prefetching, we created twelve emulated computing environments. We tested MHETA's accuracy by running both the application and our model with identical distributions for 100, 10, 5 and 10 iterations for Jacobi, CG, Lanczos, and RNA respectively. The number of iterations was chosen to obtain comparable execution times. We give a general summary of its accuracy in Section 6.2.2. We then focus on four specific configurations as described in Table 6.1, where we vary the architecture between those where only the relative CPU power varies (*DC*, for "different CPUs"), those where only the emulated disk speeds vary (*IO*, for "I/O-induced"), and those where both vary (HY,

Name	**Description**
DC	Two nodes have a lower relative CPU power, while two other nodes have higher relative CPU power. The rest are unchanged.
IO	Half of the nodes have high I/O latency with small memories, but all nodes have equal relative CPU powers.
HY1	Four nodes have varying relative CPU powers and the other four have low I/O latencies and small memories.
HY2	Four nodes have varying relative CPU power and two nodes have high I/O latencies. The other two have large memories.

Table 6.1: The configurations of the emulated architectures on which we tested MHETA.

for "hybrid configurations"). The two hybrid configurations (*HY1* and *HY2*) highlight the trade-off between balancing load and bringing the data in core. An algorithm searching for a data distribution between `I-C` and `I-C/Bal` can use MHETA to determine which point results in the lowest execution time. A discussion of the model's limitations follows in Section 6.2.3.

6.2.2 COMPARISON OF PREDICTED AND ACTUAL EXECUTION TIMES

Below, we first present the overall numbers averaged over all the different architectures we emulated. Then, we discuss the four specific architectures described in Table 6.1.

OVERALL RESULTS

The results of comparing MHETA predicted execution times and the actual run times are shown in Figure 6.2. These graphs show the maximum, average, and minimum percentage difference between MHETA, where the percentage is computed as the absolute difference divided by the minimum of each application's predicted and actual execution times. The top left graph shows that in all seventeen emulated architectures, MHETA is on average 98% accurate for all four programs without prefetching. The top right graph shows the accuracy of our model for Jacobi with prefetching in a subset of twelve architectures, also at on average 98%. The bottom graphs show the average best-case (RNA) and worst-case (CG) examples. When MHETA performs poorly, it is

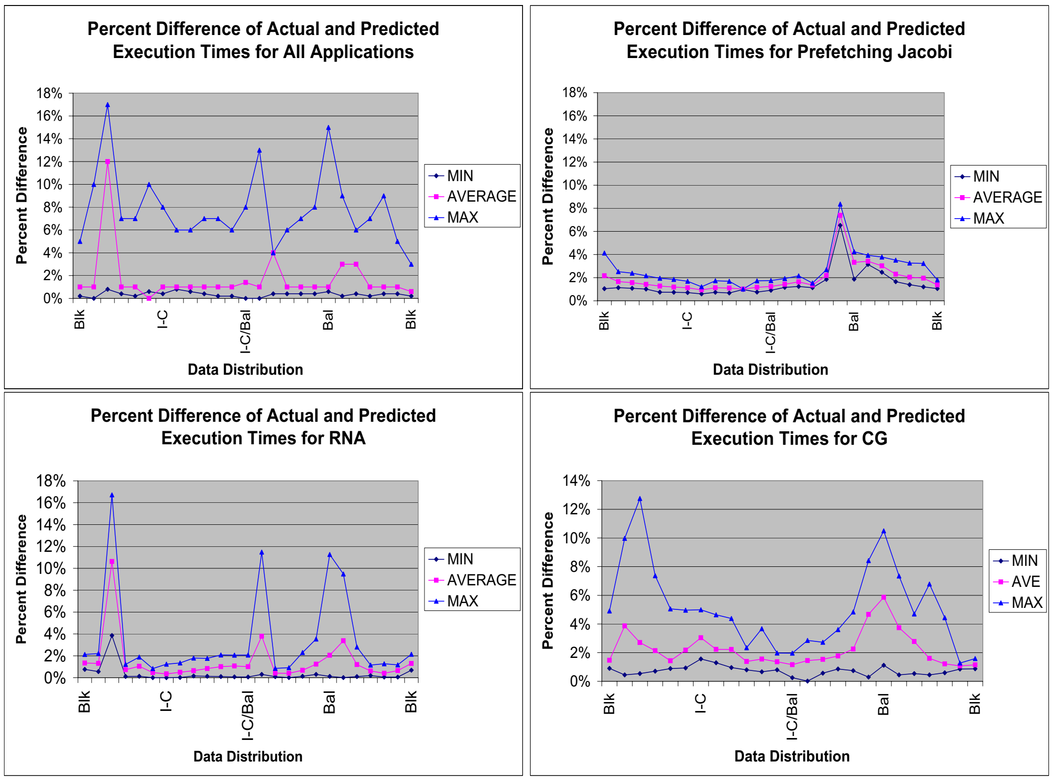


Figure 6.2: The minimum, average and maximum percentage difference between the predicted and actual execution times for all applications, and then specifically Jacobi with prefetching, RNA, and then CG. We added lines between the discrete data points for ease of readability.

in general due to three inherent limitations of our modeling approach, discussed in Section 6.2.3 below. However, even with unexpected results given these limitations, MHETA still is in general very accurate. These results show that MHETA for use by algorithms that find efficient data distributions.

Specific Architectures

Figures 6.3 and 6.4 show results using configurations *DC*, *IO*, *HY1* and *HY2*. The graphs show both predicted and actual execution times in seconds. We see in Figure 6.3 that MHETA performed very well predicting execution times of all four applications in configuration *DC*, and when predicting

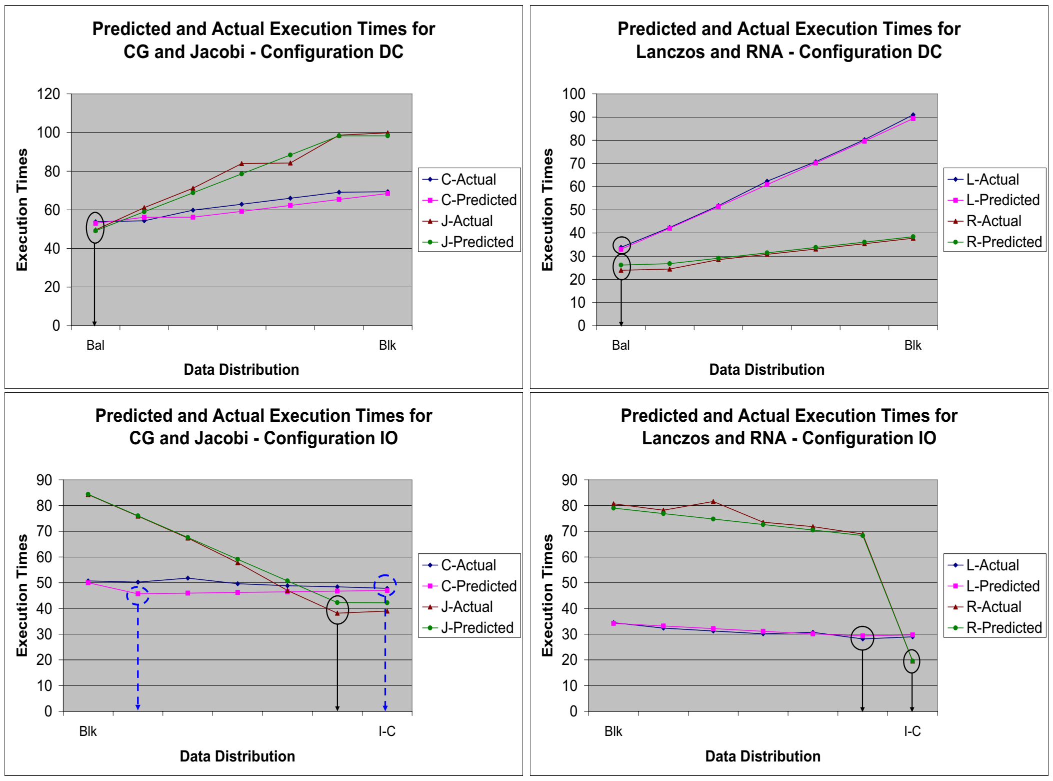


Figure 6.3: Actual vs. predicted execution times for configurations *IO* (top) and *DC* (bottom) for all four applications. The left graphs show times for Jacobi and CG, while the right show times for Lanczos and RNA. The best distributions in the set investigated for each are circled; where they match, there is only one circle, and where they do not, the predicted time has a dashed circle.

Jacobi, Lanczos, and RNA execution times in *IO*. While our model slightly overestimated the execution time for these applications right before the `I-C` distribution in *DC*, it correctly indicates this effect diminishes as the distribution approaches `I-C`. This is due to the large computation times offsetting the smaller I/O times. MHETA did not fare as well in configuration *IO* in predicting CG's execution time as shown by the two dashed circled points in the bottom left graph; however, note the difference in execution time is only 10%. This inaccuracy is because we use a heuristic to determine if a variable is out-of-core, and MHETA has difficulties with sparse arrays for CG, as explained below. Finally, MHETA predicted all test applications accurately on *HY1* and *HY2* in Figure 6.4.

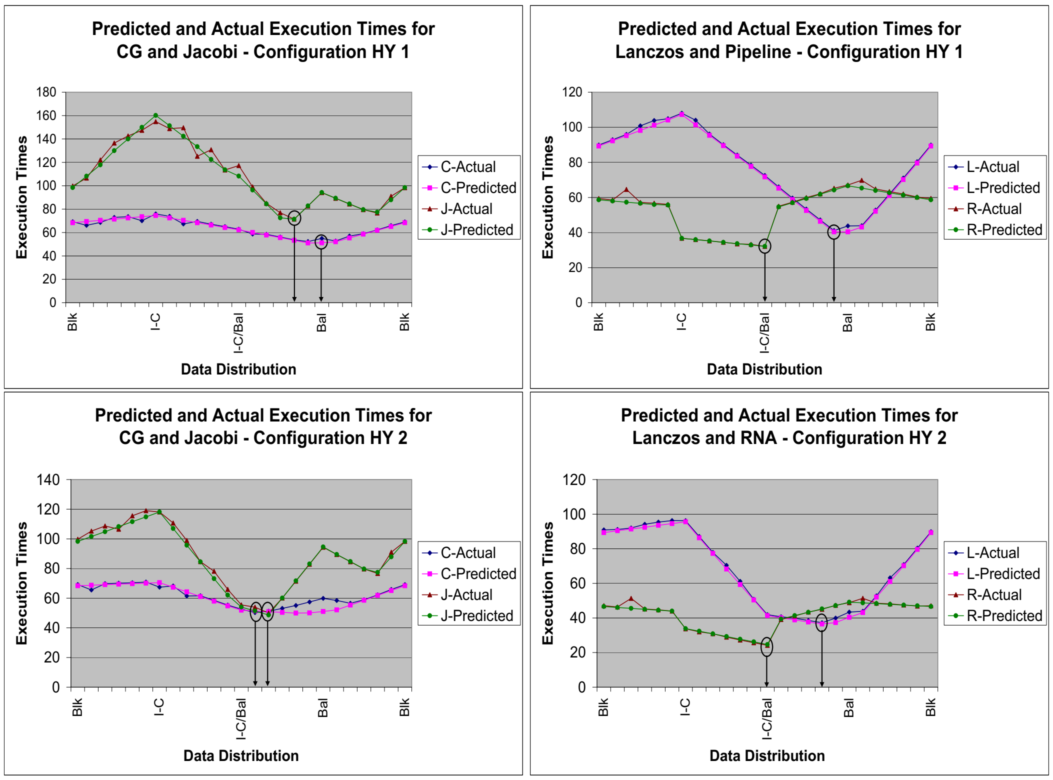


Figure 6.4: Actual versus predicted execution times for configurations *HY1* (top) and *HY2* (bottom) running Jacobi and CG (left) and Lanczos and RNA (right). The best data distribution is circled.

6.2.3 Limitations

When MHETA performs poorly, it is in general due to three inherent limitations of our modeling approach. The first limitation is that we do not model the behavior of the memory-cache hierarchy. Therefore, MHETA occasionally cannot predict when interactions with the cache and main memory cause the computation time on a node to grow or shrink. If the instrumented run records inflated or deflated computation times due to this factor, MHETA can consistently over-predict or under-predict the application execution time. Applications with higher computation costs exaggerates this problem, whereas the effects of such computation changes are not as problematic in applications where communication costs dominate execution.

The second limitation in MHETA is that the heuristic to determine which variables are out of core is not sophisticated, occasionally placing what should be an out-of-core variable in the in-core variable set. MHETA calculates the time spent performing I/O as zero when this error occurs, hence under-predicting execution time. As the data distribution shifts such that more nodes become in core, the effect of this limitation decreases. The portion of the rightmost graph from `Blk` to `I-C` distributions for MHETA shows that its accuracy for CG initially dropped from 95% in the worst case to less than 88%. As the distributions get closer to `I-C`, this difference decreased until MHETA returned to 95% accuracy, and its performance improves further as the distributions we tested shifted to `I-C/Bal`. A similar drop in accuracy occurs for the same reason with Jacobi, at the distribution right before `I-C`. Lanczos, with its higher computation to I/O ratio, minimally exhibits this problem.

Finally, MHETA (along with most data distribution systems) lacks the ability to deal with sparse datasets. Because there is not a simple correlation between number of rows and number of elements per row, there are slight load imbalances in CG that our model did not predict. We have found that despite these problems, MHETA is very effective in practice.

6.2.4 Importance of Selecting Data Distributions

This section discusses the importance of choosing a data distribution on a heterogeneous cluster. Figures 6.3 and 6.4 show that the difference between the best and worst distribution is almost a factor of 4 (RNA on *DC*) and 3 (Lanczos on *HY1*). While on a simpler architecture such as *DC*, it may be possible to determine which distribution is best statically (if one knew the relative powers of each processor), in general this determination is nontrivial. For example, due to the low I/O costs and relatively high load imbalances in configuration *HY1*, one would expect the best data distribution to be `Bal`. While this is the case for Lanczos, the best distribution for Jacobi is instead between `I-C/Bal` and `Bal`. Importantly, this distribution is *significantly* better (28%) than `Bal`. Furthermore, for different architectures that have varied I/O costs and load imbalances on the nodes, it becomes harder to predict which distribution will be best—meaning that a "guess" may end up far from the best choice, which as stated above can result in a doubling or tripling of execution time. MHETA is able to accurately predict execution times on average 98% of the time, so it can be an effective tool to use when searching for effective distributions. The basic idea is that any reasonable algorithm for choosing a data distribution on a heterogeneous cluster will need as a building block the ability to accurately predict execution time for different distributions. Section 6.3 describes different data distribution search algorithms that use MHETA during their search process; (genetic algorithms, simulated annealing, generalized binary search, and greedy search)

6.3 Performance Analysis of the Search Algorithms

We investigate the relative and overall effectiveness of the four search algorithms detailed in Chapter 4—the *GBS* and *GS* algorithms that are specifically tailored to this problem and the *GA* and *SA* algorithms that are more general in nature. First, we tested the algorithms on eight nodes, emulating a variety of heterogeneous architectures as described in Section 6.1. We ran all four algorithms for one second to keep the overhead of using the SHARED system tolerably low for the user. Our results show that after running for this period of time, both *GBS* and *GS* on average produce a data distribution that is 13% better than either *SA* or *GA*. They also are within 5% of the *Best* distribution (see below). The similarity of *GBS* and *GS*'s performance is primarily due to (1) their technique of directly reducing the component of a distribution that results in the maximum execution time and (2) their low calculation overheads. Secondly, we tested the scalability of the *GBS* algorithm versus *GS* on Lanczos and Jacobi running on an simulated architecture of 16, 32, 64, and 128 nodes. *GBS* consistently produces better data distributions than the *GS* algorithm for both applications running on architectures with up to 32 nodes. When the number of nodes of the target architecture increases to 64 and 128, however, *GS* initially finds better distributions because it scales better than *GBS*. For each data distribution vector found, *GS* performs a single scan through the elements to improve the distribution, whereas the *GBS* algorithm sorts them. For

Lanczos running on 128 nodes, *GBS* is thus slower than *GS* in finding a solution until 22 seconds have elapsed; for Jacobi, this point is reached in 0.5 seconds.

Section 6.3.1 describes the setup specific to these sets of tests. Next, Section 6.3.2 gives composite performance over all applications and several different heterogeneous architectures. Section 6.3.3 contains an in-depth comparison of a single program on two specific architectures, followed by an evaluation of the scalability of our optimal *GBS* algorithm versus *GS* in Section 6.3.4.

6.3.1 Setup Specific in Testing Search Algorithm Performance

We intend to eventually use the best of our four algorithms in our SHARED system to make on-the-fly changes in data distribution, so our algorithm will have to run within a short period of time, as detailed in Sections 6.3.2 and 6.3.3. When we compared how quickly *GBS* and *GS* produce effective data distributions as the number of nodes in the parallel architecture increases, initial tests show no further changes resulted after running for more than 30 seconds. Therefore, we ran both algorithms for a maximum of 30 seconds in the tests detailed in Section 6.3.4.

Each of the four algorithms output the best data distribution they could find. We also have a distribution we call *Best*, which is output by the *GBS* algorithm when it completes. Unfortunately, there are two problems with this technique of calculating *Best*. First, as the number of nodes in the input distribution vector increases, the *GBS* algorithm takes longer to finish. This technique may therefore take longer than suitable for a runtime data distribution system, but it serves as a way to evaluate our algorithms. Secondly, *GBS* does not factor blocked communication time into its search, as explained in Chapter 4, so it does not produce an optimal distribution when the application spends a significant portion of its execution blocked waiting for messages.. In those cases, the *Best* distribution (or some close approximation) is found by hand (described in more detail in Section 6.3.3).

The *GA*'s effectiveness in avoiding local optima depends on correct initial conditions, which are difficult to establish *a priori*—they are specific to the application and target architecture for which we are searching for a distribution. After some initial tests, we determined that a population

Name	Description
DC	Two nodes have a lower relative CPU power, while two other nodes have higher relative CPU power. The rest are unchanged.
IO	Half of the nodes have high I/O latency with small memories, but all nodes have equal relative CPU powers.
HY1	Four nodes have varying relative CPU powers and the other four have low I/O latencies and small memories.
HY2	Four nodes have varying relative CPU power and two nodes have high I/O latencies. The other two have large memories.
HY3	**Six nodes have varying relative CPU power and three nodes have varying I/O latencies. Note that some nodes will have both a slow disk and low CPU power.**

Table 6.2: The configurations of five of the emulated architectures on which we tested the performance of *GBS*, *GS*, *GA* and *SA* algorithms. Note that the *HY3* configuration is new to these set of tests.

of 100 candidate solutions and a mutation probability of 5% help prevent it from converging to a local optimum. All the algorithms started with a random data distribution (in the case of the *GA*, it was 100 random starting points). Due to the high overhead of maintaining and creating large populations, however, the *GA* still converges on a solution fairly slowly.

We used 21 different emulated computing environments per application. A majority of these emulated architectures are the same as when we tested MHETA's accuracy, and out of the five we examine in detail, four have already been detailed in Section 6.2.1. The additional architecture, *HY3*, has six nodes with varying CPU power and three nodes with varying I/O latencies. Note that some nodes have both a slow disk and low CPU power. Table 6.2 characterizes the five emulated architectures we detail in the following sections. For completeness, we also have a set of architectures that emulate a slower CPU on a random subset of nodes and a smaller memory on a random (and possibly intersecting) subset of nodes. The intention is to mimic an arbitrary cluster.

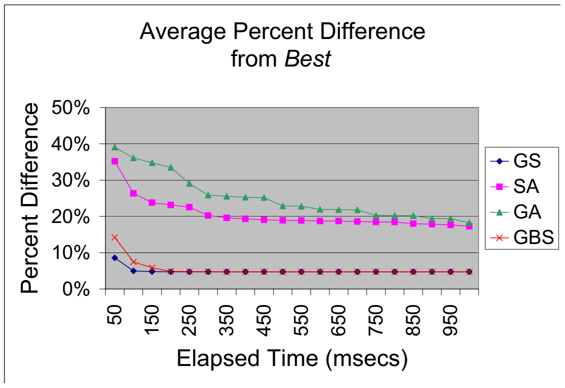


Figure 6.5: The percentage difference between *GS*, *GA*, *SA*, and *GBS* algorithms from *Best*. Tests are run over a set of five emulated architectures and the Jacobi, CG, and Lanczos applications.

6.3.2 Overall Results

In this section, we present overall results for several different architectures and the three applications, Jacobi, CG, and Lanczos. We compare the execution time of the application given the best distribution discovered by *GBS*, *GA*, *SA*, and *GS* with the time if the application is run with the *Best* distribution. Figure 6.5 shows the percentage difference between these times over all architectures listed in Table 6.2. We plot the result produced by each algorithm every 50 ms until one second is reached. A 0% difference means an identical result to *Best*. *SA* performs at 40% of *Best* at 50 ms, primarily due to the fact that during the melting process, the solution is in a very chaotic state. However, it steadily improves, reaching within 19% of *Best* at one second. A negative slope suggests that further improvement is likely, as the cooling rate is sufficiently slow to prevent it from getting "stuck" at a local minimum. The *GA* is also within 39% of *Best* by 50 ms. The *GA* saves its best candidates from one generation into another, but has a tendency to allow the population to converge at a local minimum, which is offset by the mutation factor. At the end of one second, both *SA* and *GA* are 7% of *Best*. Both *GBS* and *GS* are 0% of *Best* by one second in general, but neither algorithm reaches *Best* for the CG application due to the fact that blocked time is a major part of communication for this program.

6.3.3 Detailed Results

This section focuses on an in-depth comparison of performance of all algorithms on two specific application/architecture pairings: Lanczos on *DC* and CG on *HY2*. For each pair, we investigate three different results. We first show in detail how different each search algorithm is from *Best*. Second, we show the difference in predicted execution time (per iteration). Third, we show that MHETA is accurate, which verifies that the best distributions found by these algorithms correspond to effective distributions for the application actually running on the architecture.

The results are shown in Figure 6.6. The high computation to communication ratio for Lanczos results in significant differences in execution time for different data distributions. As the top left graph shows, at 200 ms, both *GS* and *GBS* are at 0% of *Best* and 78% better than *GA*. The performance of the *GA* further improves to 17% of *Best* after 1 second. *SA* initially performs poorly within 80% of *Best* at 50 ms, but it quickly improves to within 20% of *Best*by 150 ms. There is subsequently only a gradual improvement in the solution as it starts from random points in the search space. The execution time difference, shown in the top center graph, is small in an absolute sense between *GA* (or *SA*) and *GBS* (or *GS*) (4 seconds); however, note that this is the difference for *each* iteration. In addition, *GBS* and *GS* are within 5% of the lower bound, which is the execution time of the application *without* considering communication blocked time. Finally, our results verify our approach to model the execution of these applications in searching for a data distribution. There is a small negative difference between the predicted and actual execution times (shown in the top rightmost graph).

Due to the low ratio of computation to communication in CG, changes in the data distribution do not greatly effect execution time. After running the algorithms for 50 ms, none improve, as shown in the bottom left graph. Of some interest is that now *SA* outperforms *GA* by at least 35%. In addition, this low ratio highlights the problem that ignoring communication blocked time can

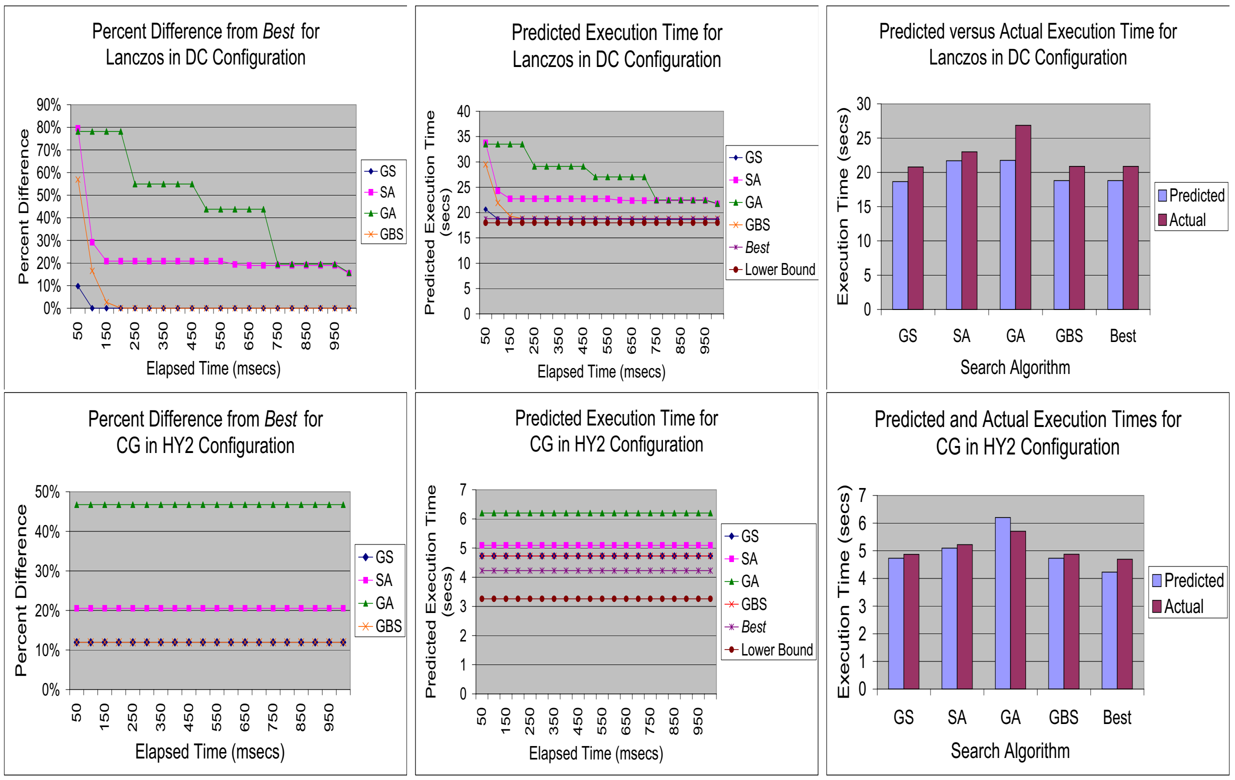
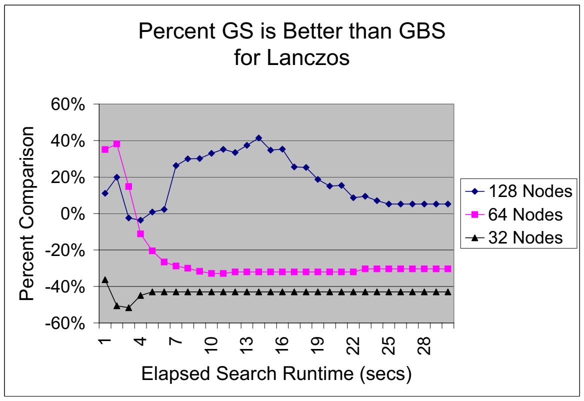


Figure 6.6: The performance of the heuristics on the *DC* configuration for Lanczos and the *IO* configuration for CG. Note that *GBS* and *GS* times overlap in the middle graphs. The predicted and actual execution times are per iteration.

present to *GBS* and *GS*—both are at least 20% worse than *Best*, which we found by hand. This point is further highlighted in the bottom center graph. The lower bound is far less than *Best* (1.7 seconds, or 50%), indicating that blocked communication time is a significant portion of the total execution time. Again, the bottom right graph shows that the predicted execution times are close in practice to the actual execution times.

6.3.4 Scalability

We compare the performance of *GBS* and *GS* over both Lanczos and Jacobi programs running on a simulated architecture of 32, 64, and 128 nodes. We ignore the *GA* and *SA* algorithms, as



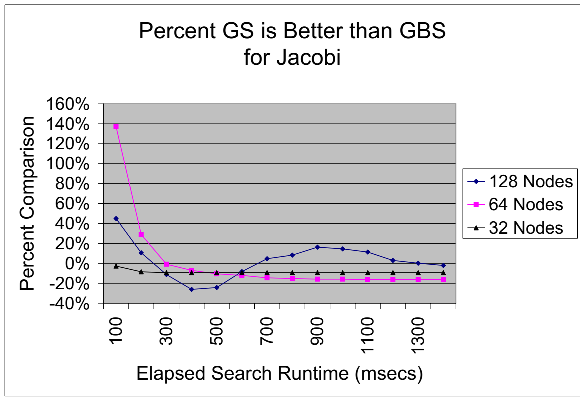


Figure 6.7: The scalability performance of the *GBS* and *GS* algorithms for Jacobi and Lanczos.

they do not scale well at all. The *GA* must successively generate new populations, which is a very slow process at larger numbers of nodes, and the "cooling" of the search vector in the *SA* algorithm also becomes very slow. The results of Lanczos and Jacobi are the left and right graphs in Figure 6.7. At larger number of nodes, the drawback of the *GS* algorithm becomes evident; i.e. because it does not do a pairwise examination of all values, it gets stuck in a local optima. The *GS* algorithm thus consistently performed worse than *GBS* for both applications at 32 nodes, as shown by the negative values in the graphs. At large numbers, however, the faster speed of *GS* enables it to initially outperform *GBS*. However, after 3 seconds for Lanczos and 300 ms for Jacobi, this advantage again disappears, and *GBS* is the best algorithm for 64 nodes. After a period of time, the *GS* algorithm again performs well—as the *GBS* algorithm slows down to find the optimal solution, the greedy search proceeds to a better solution faster. This advantage quickly disappears again for Jacobi (within a second), and much more gradually in Lanczos (22 seconds).

6.3.5 Summary of Results

We draw the following conclusions from our results. First, *GBS* and *GS* are typically more effective search algorithms in general for the out-of-core, heterogeneous data distribution problem than *SA* or *GA*. In particular, with moderate to high computation to communication ratios, and small numbers of nodes, *GBS* and *GS* find distributions that are 1.5 three times faster than *GA* or *SA*. However, with low computation to communication ratios, *SA* and *GA* may be better, but not by nearly as significant an amount. Second, as the number of nodes increases, *GS* and *GBS* scale much better than *SA* or *GA*. Between *GS* and *GBS*, the former is better for (at most) the first few seconds, while the latter dominates from that point on. The proper algorithm to use depends on how long the runtime system is willing to search for a data distribution.

6.4 General Conclusions

Our results show that our SHARED system should use the *GBS* algorithm with the MHETA evaluation function when producing effective data distributions. This conclusion is valid when the communication blocked time is not a significant portion of an application's total execution time (its computation or I/O to communication ratio is high).

First, we determine that our MHETA model is a suitable and useful evaluation function for the search algorithm in SHARED because it is at least 97% accurate in predicting execution times for applications in our testbed. Second, we show that although *GBS* and *GS* are both significantly better than *GA* and *SA* in producing effective data distributions quickly for Jacobi and Lanczos, *GBS* does not scale as well as *GS*. *GBS* consistently finds more effective distributions when there are fewer than 64 elements in the distribution vector. When this vector size increases to 64 and larger, the *GS* algorithm initially performs better because it takes less time to generate better distributions. *GS* can get stuck in a local optima, however, whereas the *GBS* algorithm can eventually find the optimal distribution given a reasonable amount of time. The *GBS* algorithm is thus the best choice , as it is comparable to *GS* when the size of the distribution vector is small and eventually out-performs *GS* with larger vector sizes.

CHAPTER 7

CONCLUSION AND FUTURE WORK

I/O costs are an important factor for two reasons when scheduling or distributing data for scientific applications. First, parallel applications are increasingly likely to run on a more cost effective heterogeneous set of “off the shelf” machines that can have potentially limited memory capacities. A data distribution onto this architecture has a higher probability of assigning more data to a node that it can hold in its memory. Second, large scale scientific applications can easily increase their working dataset size either by increasing resolution or accuracy requirements, again exceeding the memory capacity of the underlying architecture. The assumption of infinite memory is therefore less feasible given these factors, and mechanisms to assess the costs to access the disk must be included in runtime systems for parallel applications.

After verifying through a gang scheduling simulator the importance of I/O-awareness in scheduling, we formalized the out-of-core, heterogeneous data distribution problem. A brute force search for the most effective distribution is computationally intractable, and there are two methods to solve this problem adequately. The first is to simplify the problem; we assume monotonicity between a data distribution and the resulting execution time of an application and ignore communication blocked time. The *GBS* algorithm is optimal with this simplification and finds a solution in polynomially bound time. The second is to employ a heuristic to find near-optimal data distributions, such as simulated annealing (*SA*), genetic algorithms (*GA*) or greedy search (*GS*). Note that *GBS* does not produce an optimal distribution if it either (1) does not run to completion or (2) communication blocked time is a significant portion of an application’s execution time. When the number of elements in the distribution vector is large, *GBS* takes longer to finish (as does all the other algorithms), and it is less likely to produce the optimal distribution within the time limit.

We studied the relative and absolute performance of *GBS* versus these other three heuristics in finding an effective data distribution within one second of execution. Suppose there are N nodes, an initial distribution vector $x = \{x_0, \ldots, x_N\}$ and function $f_i(x_i) = g_i(x_i) + h_i(x_i)$, where g_i and h_i return the duration for the i^{th} node to perform computation and I/O, respectively. For each, we compared the distributions output by all four algorithms to (1) the optimal distribution from running *GBS* to completion as well as (2) the lower bound on application execution time, which is $= \min_{i=0\ldots N} f_i(x_i)$.

Results show that in the majority of cases, the *GS* and *GBS* algorithms are very effective. When running both of these for one second, they were on average within 2% of the best distribution for four applications running on a variety of emulated architecture. *GS* and *GBS* are better than a *GA* because they sample points in the search space quickly without the overhead of maintaining and processing multiple candidates. They also do not rely on any initial conditions, whereas an *SA*'s performance depends on the best cooling schedule, which is often difficult to determine *a priori*. For the CG application, however, *GBS* is not as effective because CG spends a significant amount of time in blocked communication. When comparing how well *GBS* and *GS* scale to the number of elements in the solution vector, we find that *GS* is initially more effective than *GBS* when the number of nodes in the architecture is small. Unlike the *GBS* algorithm, *GS* does not perform any sorting, so given equal amounts of time, it explores more points in the search space than *GBS* and finds better solutions quickly. However, the *GBS* algorithm is still comparable when the vector size is small, and once the number of elements increases, the *GS* algorithm is likely to get stuck in a local optimum, whereas the *GBS* algorithm proceeds to find the optimal distribution.

All the search algorithms require an evaluation function for each candidate distribution, and we developed an accurate computation model called MHETA for this purpose. MHETA consists of a system of equations that include the costs of computation, communication and I/O specific to the application running on the target architecture in its prediction of its execution time. Our model is implemented such that the equation structure is built from information, extracted by a pre-processor, about the relationship between an application's parallel sections, stages, tiles and

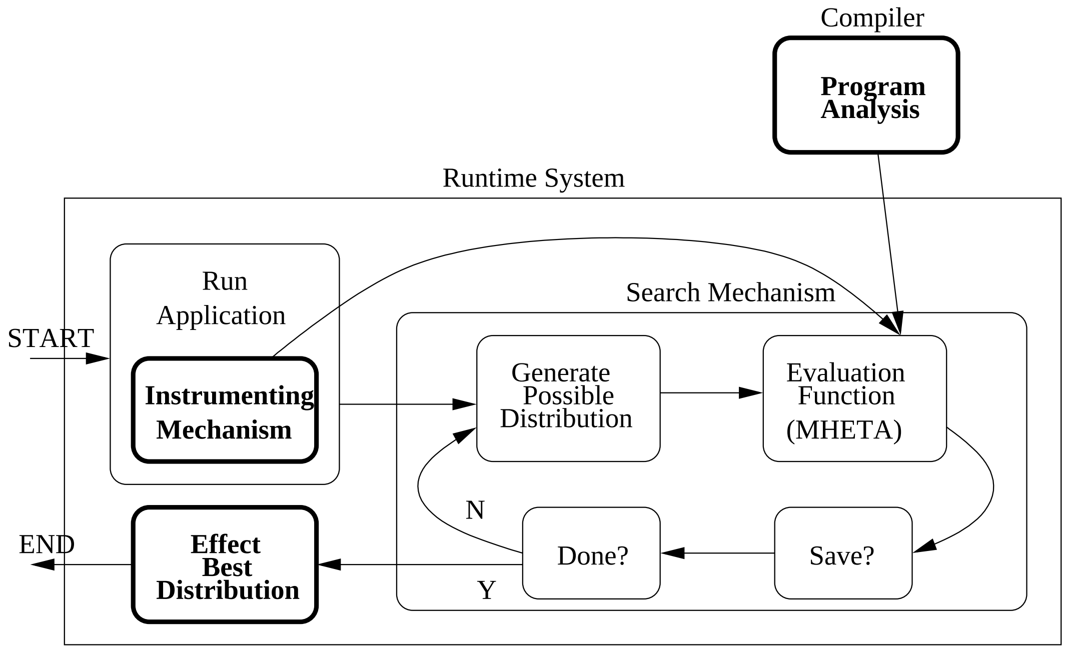


Figure 7.1: The general structure of the SHARED runtime system. The **bold** components are for future work.

the variables they use. We target iterative applications, and the values for the parameters of these equations is measured during an instrumented iteration run with an arbitrary distribution. Experiments with Jacobi with and without prefetching, CG, Lanczos and RNA show MHETA predicts their execution time with on average at least 96% accuracy. Therefore, *GBS* (along with MHETA) will serve as a key component of a runtime system we are developing that can select an effective distribution on the fly.

7.1 Future Work

For the short term, I will continue developing this runtime system, shown in Figure 7.1, by enhancing certain components and adding others. I plan to enhance the evaluation function to increase the types of architectures and communication patterns that this system can accommodate when searching for an effective data distribution. For example, MHETA can easily be expanded to correctly calculate the costs to access remote disks. However, the system must then be able to detect which I/O operation is invoked to use the correct equations. In addition, more research is required to factor how network congestion can cause increased delays beyond what can be predicted when accessing remote disks. MHETA can also be enhanced to calculate the costs of performing more communication patterns, such as all-to-all, shuffle, and cascade message passing.

I intend to design and implement the component that automatically extracts application structure information, which is required by MHETA and is currently done by hand. A compiler has access to the program source code, so it can view global relationships between instructions and split the program into the parts that each equation in MHETA represents. My focus is on parallel and distributed systems, so the algorithm I already formalized to perform this is relatively simple—it does not perform any inter-procedural analysis, for example. I will further the design so that it can perform more complex analysis.

With regard to broader issues, I plan to investigate other important aspects of parallel and distributed computing, such as scheduling, data redistribution, and task parallelism. I have already shown through simulation that gang schedulers could benefit from knowledge of an application's I/O behavior, and I plan to examine how other coordinated kinds of scheduling could be designed to minimize the impact of I/O costs on execution times. In environments where resources must be shared, runtime systems such as Dyn-MPI [58] typically *redistribute* data. Building on the ideas I contributed in [58], I would like to investigate how I/O costs can be integrated with these systems. For example, they could use I/O costs in cost/benefit analysis to effectively determine *when* to perform the redistribution. Finally, certain phases of scientific programs could operate more efficiently when their individual tasks run in parallel (task parallelism). The runtime system could thus use I/O behavior information in determining whether to perform task versus data parallelism if both are possible, and when to switch from one to the other.

This research is the first step in developing our SHARED system that automatically maps work of a parallel application onto the underlying architecture that will minimize execution time. Although we implemented several key components, various stages have yet to be completed. User still must analyze the application source code by hand and insert some instrumenting code to be used by SHARED later, which is a tedious and error-prone process. In addition, they then must manually run the application with the distribution found by *GBS*. With further work and enhancements, our system will free the user of this burden. It will take a parallel scientific application and automatically (1) analyze its structure, (2) measure costs specific to its execution during an instrumented run, (3) construct an accurate model and (4) search for an effective data distribution. It will then take the distribution output by the search algorithm and run the program with the new data mapping based on that distribution. Our system correctly includes I/O costs, and is thereby applicable in more situations than traditional approaches that assume infinite memory capacities and are I/O oblivious.

BIBLIOGRAPHY

[1] Impie: Interposed message passing interface executor. North Carolina State University website: http://fortknox.csc.ncsu.edu/proj/impie/.

[2] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 305–314. ACM Press, 1987.

[3] G. Agrawal, A. Acharya, and J. Saltz. An interprocedural framework for placement of asynchronous I/O operations. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 358–365, Philadelphia, PA, 1996. ACM Press.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[5] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Proc. Programming Models for Massively Parallel Computers*, 1993.

[6] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 93.

[7] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed hetergeneous networks. In *Proceedings of Supercomputing '96*, page 39. ACM Press, 1996.

[8] H. Bodhanwala, L. M. Campos, C. Chai, C. Decoro, K. Fowler, P. Franck, H. Nguyen, N. Patel, I. Scherson, and F. Silva. A general purpose discrete event simulator, July 2001.

[9] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Santa Barbara, CA, 1995. ACM Press.

[10] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and evaluation of primitives for Parallel I/O. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 452–461. ACM Press, 1993.

[11] K. K. Cameron and X.-H. Sun. Quantifying locality effect in data acess delay: Memory logP. In *International Parallel and Distributed Processing Symposium*, pages 48–55, Apr. 2003.

[12] E. Caron, O. Cozette, D. Lazure, and G. Utard. Virtual memory management in data parallel applications. In *HPCN Europe*, pages 1107–1116, 1999.

[13] F. W. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. In *Operating Systems Design and Implementation*, pages 1–14, 1999.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms: Second Edition*. MIT Press, 2001. pages 1096–1097.

[15] O. Cozette, A. Guermouche, and G. Utard. Adaptive paging for a multifrontal solver. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 267–276. ACM Press, 2004.

[16] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[17] D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In *IPPS 1997 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 238–261, Apr. 1997.

[18] D. G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Journal of Parallel and Distributed Computing*, 23(5):65–77, May 1990.

[19] D. G. Feitelson and L. Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[20] M. Fleischer. Simulated annealing: Past, present, and future. In *Winter Simulation Conference*, pages 155–161, 1995.

[21] I. Foster and N. T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98*. ACM Press, 1998.

[22] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[23] M. Grigni and F. Manne. On the complexity of generalized block distribution. In *IRREGULAR: International Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117 of *lncs*, pages 319–326, 1996.

[24] M. Gupta and P. Banerjee. PARADIGM: A compiler for automated data distribution on multicomputers. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 357–367, July 1993.

[25] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, Houston, Tex., Nov. 1994.

[26] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua,

editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop,* Santa Clara, CA, 1991. Springer-Verlag.

[27] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communication of the ACM*, 35(8):66–80, Aug. 1992.

[28] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Run-time and language support for compiling adaptive irregular programs on distributed-memory machines. *Software - Practice and Experience*, 25(6):597–621, 1995.

[29] J. F. J. J and K. W. Ryu. The block distributed memory model. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):830–840, 1996.

[30] M. A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of Supercomputing*, Nov. 1997.

[31] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[32] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, 1994.

[33] D. Kotz and C. S. Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 182–189. IEEE Computer Society Press, 1991.

[34] W. Lee, M. Frank, V. Lee, K. Mackenzi, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[35] X. Ma, M. Winslett, J. Lee, and S. Yu. Faster collective output through active buffering. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 151. IEEE Computer Society, 2002.

[36] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 61. IEEE Computer Society, 1995.

[37] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, pages 1087–1092, 1953.

[38] D. G. Morris and D. K. Lowenthal. Accurately computing redistribution cost in distributed shared memory systems. In *Principles and Practice of Parallel Programming*, pages 62–71, June 2001.

[39] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, October 1996.

[40] M. P. I. F. MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, Tennessee, 1996.

[41] N. J. Nevin. The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputing Center, Columbus, Ohio, 1996.

[42] D. S. Nikolopoulos and C. D. Polychronopoulos. Adaptive scheduling under memory pressure on multiprogrammed clusters. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 22. IEEE Computer Society, 2002.

[43] D. J. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1995.

[44] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, New York, NY, 1995.

[45] J. Ramanujam and A. Narayan. Automatic data mapping and program transformations. In *Workshop on Automatic Data Layout and Performance Prediction*, June 1995.

[46] U. Rencuzogullari and S. Dwardadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. *ACM SIGPLAN Notices*, 36(7):72–81, 2001.

[47] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, 1995. IEEE Computer Society Press.

[48] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 26(7):42–50, July 1993.

[49] P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Lecture Notes in Computer Science, Workshop on Parallel Job Scheduling, IPPS '95*, pages 106–126. Springer Verlag, 1995.

[50] M. S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *Job Scheduling Strategies for Parallel Processing*, pages 219–238. Springer Verlag, 1995.

[51] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–319, Dec. 1990.

[52] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and runtime support for out-of-core HPF programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, Manchester, UK, July 1994. ACM Press.

[53] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, 1994.

[54] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. SMARTS: exploiting temporal locality and parallelism through vertical execution. In *Proceedings of the 13th International Conference on Supercomputing*, pages 302–310. ACM Press, 1999.

[55] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[56] D. E. Vengroff and J. S. Vitter. I/O efficient scientific computation using TPIE. Technical Report TR–1995–18, Duke, 1995.

[57] J. S. Vitter and E. A. M. Shriver. Algorithms for Parallel Memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[58] D. B. Weatherly, D. K. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters. In *Proceedings of Supercomputing*, Nov. 2003.

[59] G. E. Weaver, K. M. McKinley, and C. C. Weems. Score: A compiler representation for hetergeneous systems. In *The Hetergeneous Computing Workshop*, pages 10–23, Apr. 1996.

[60] T. L. Williams and R. J. Parsons. The heterogeneous bulk synchronous parallel model. *Lecture Notes in Computer Science*, 1800:102–??, 2000.

[61] T. Yeh, D. D. E. Long, and S. Brandt. Performing file prediction with a program-based successor model. In *MASCOTS 2001*, pages 193–202, Cincinnati, OH, Aug. 2001. IEEE Computer Society.

APPENDIX A

COMPLEX ANALYSIS OF SCIENTIFIC APPLICATIONS

The pre-processor that defines the parallel sections and stages in an application as described in Chapter 5 operated under several simplifying assumptions. The states and transitions in the deterministic finite automaton of Figure 5.3 sufficed for programs that (1) have stages explicitly defined by loop bounds and (2) did not allow nested loops in a stage. Scientific programs are generally more complex with complicated stages, and we first detail changes to the pre-processor that allows it to correctly determine stages. We follow with further enhancements for the the correct detection of tiles.

The transitions and states needed for this more complex analysis of parallel sections and stages are in the *nondeterministic* finite automaton (NFA) shown in Figure A.1. To allow for a stage to implicitly start with a parallel section, there is an additional transition from the Parallel Section to Stage states taken when the current instruction is neither the start of a loop (loop_start) nor a communication event (Comm). Encountering Comm while in a stage indicates the end of that stage and the parallel section, so the $next$ variable is set to true during the transition out of the Stage state. Once in the Parallel Section state, the transition to increment the $section$ variable is taken immediately.

Nested loops within a stage are handled through the addition of two states. The Loop Start and Loop End states are entered when the instructions indicating the start (loop_start) and end (loop_end) of a loop are encountered, respectively. (The ϵ transition from Loop Start to Stage is given a condition when we expand the NFA to include tiles.) We use the variable $\#loops$ to control when the outermost loop of the stage finishes (i.e. $\#loops = 0$). A stage that started implicitly with

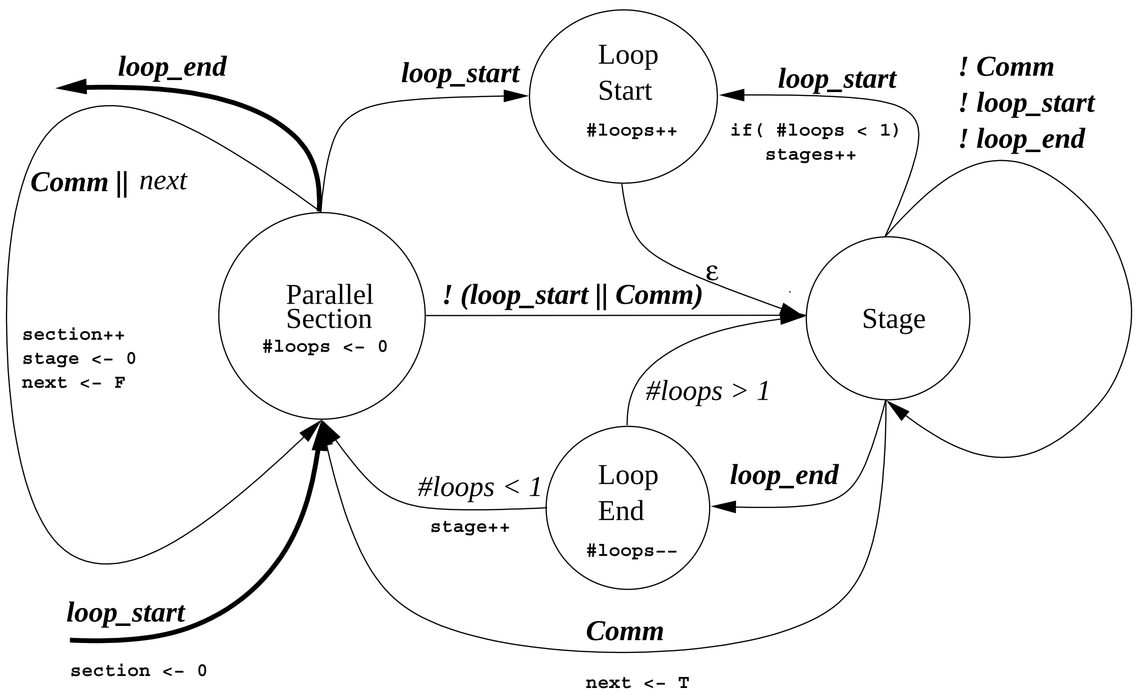


Figure A.1: A directed graph expands the diagram in Figure 5.3 to include nested loops and stages boundaries that are implicit with parallel sections. The ***italic and bold*** terms are statements (tokens) encountered in the source code. *Italicized* terms are conditions that determine whether the transition occurs. `Courier and bold` terms are actions for internal variables.

a parallel section can end and another one start when a loop is encountered, so the transition from the Stage to Loop Start states have a conditional increment of the $stage$ variable.

The state diagram becomes even more complex when determining where tiles start and end because a loop can now indicate the existence of *both* stages and tiles. However, stages can have nested loops, whereas tiles cannot. This new NFA that distinguishes between stages and tiles is shown in Figure A.2. There are three new states (Tile Start, Tile End, and Comm Event) and multiple new transitions that are taken under specific conditions. A typical implementation of pipelining involves a potential receive for boundary information, followed by computation over local data, and ending with potentially sending data to another node. A tile is thus bounded by the start and end of a receive-compute-send loop. The start of a tile is indicated by detection of

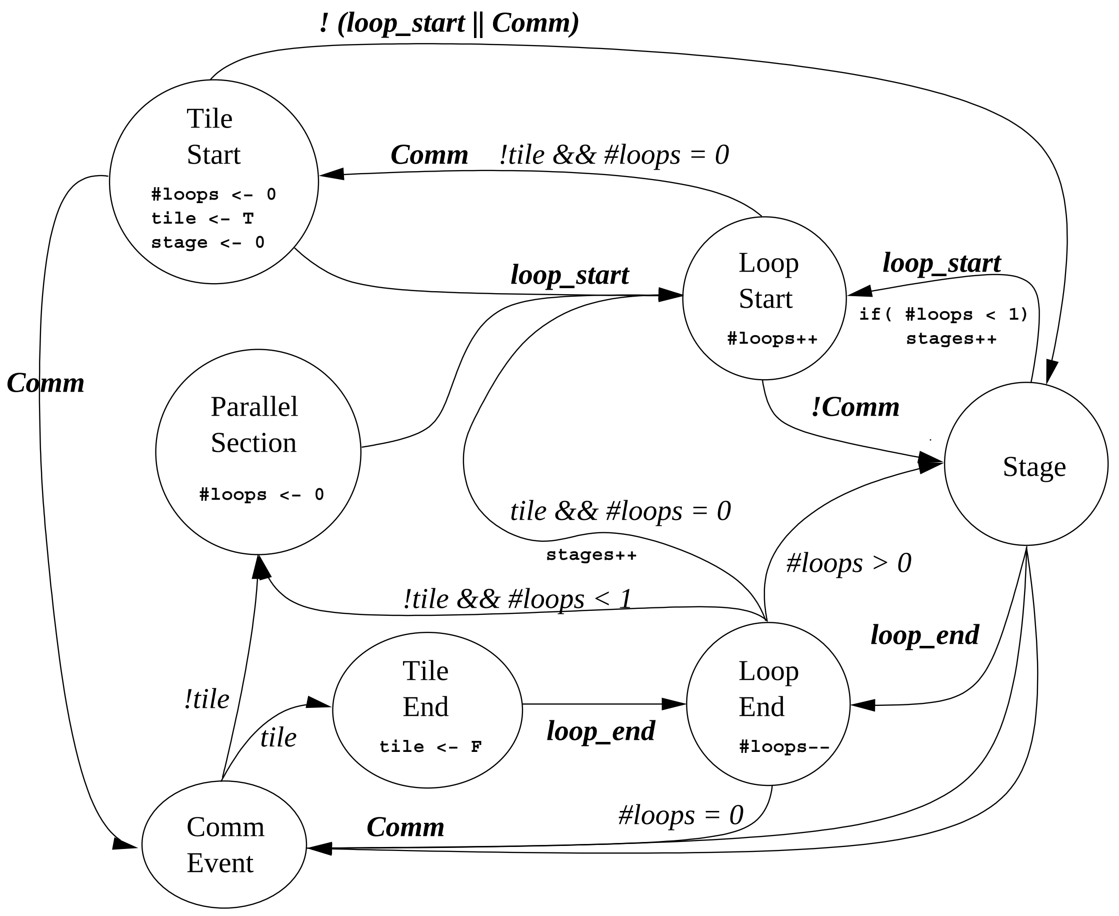


Figure A.2: A directed graph expands the diagram in Figure A.1 to include identifying tiles. Several edges have been removed for legibility, and unmarked edges indicate that the conditions for leaving a state has not changed from the previous diagram.

loop_end followed immediately by Comm; otherwise, it enters the Stage state. In the Tile Start state, $\#loops \leftarrow 0$ to correctly detect nested loops in the stages in the tile, sets the variable $tile$ to true to control which transition is taken when in the Comm Event state, and sets the number of stages to zero. If another Comm immediately follows, the tile is assumed to have no computation, and because the variable $tile$ is set to true, we proceed to the Tile End state. Alternatively, the pre-processor enters the Stage state either by encountering loop_start (after which there could be nested loops in a stage), or if a stage is implicitly as bounded by the tile itself. If Comm is encountered while in a stage, either the pre-processor enters the Parallel Section or Tile End based on whether $tile$ is false or true, respectively. If Loop Start is encountered after a stage ends while within a tile, the pre-processor enters the Loop Start state for another stage. A loop_end encountered while at the Tile End state returns the pre-processor back to the Parallel Section state.