APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for

the degree of Doctor of Philosophy in Computer Science

Jon B. Weissman

This dissertation has been read and approved by the Examining Committee:

Dissertation Advisor: Andrew Grimshaw

Committee Chair: William A. Wulf

Committee Member: James Ortega

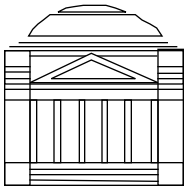
Committee Member: Paul Reynolds

Minor Representative: James Aylor

Accepted for the School of Engineering and Applied Science:

Dean Miksad
School of Engineering and Applied Science
August, 1995

Scheduling Parallel Computations in a Heterogeneous Environment



A Dissertation presented to

the Faculty of the School of Engineering and Applied Science

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy in Computer Science

Jon Weissman

August 1995

To my wife

Only those who will risk going too far can possibly find out how far one can go.
— T.S. Eliot

Acknowledgments

My thanks go to the many Mentat team members past and present that I have had the opportunity to work with over the years. All of you have helped build a system infrastructure from which some wonderful research has blossomed. This dissertation would not have been possible without these efforts.

My examining committee, Bill Wulf, Andrew Grimshaw, James Ortega, Paul Reynolds, and James Aylor provided a careful reading of the dissertation and made many helpful suggestions.

Special thanks go to my advisor Andrew Grimshaw who taught me that good research is based on commitment and hard work, but that great research is built on faith. His vision of a wide-area virtual computer has been an inspiration in my work. I am truly honored to be his first Ph.D. student.

Robert Ferraro and the NASA-Jet Propulsion Laboratory supported me through a GSRP research fellowship. The fellowship provided a unique opportunity to collaborate and meet with NASA scientists. This collaboration and interaction improved the quality of this dissertation greatly.

Finally, the support of my friends and family including my wife Susan, my brother Steve, and my parents, kept me feeling positive and helped me weather the tough times. My wife Susan was a constant source of motivation and understanding and it is with much love that I dedicate this dissertation to her.

Table of Contents

List of Figures

List of Tables

List of Symbols

N_i = the i^{th} network cluster

C_j = the j^{th} processor cluster

P_j = number of processors selected for C_j

P_T = total number of processors selected

τ = application communication topology

b = message size in bytes

c_1 .. c_4 = communication cost constants

f () = cluster-dependent communication function

r_1, r_2 = router cost constants

e_1 = conversion cost constant

v= number of messages that cross between each processor cluster

p_i = a particular processor

A_i = number of PDUs assigned to processor p_i

V_j = number of available processors within cluster C_j

w_i = relative processor weight for i^{th} processor (problem-specific)

m = number of clusters

g() = the amount of computation as a function of A_i

x_i = PDU independent cost constant for i^{th} processor

y_i = PDU dependent cost constant for i^{th} processor

T_c = per cycle elapsed time

DP = set of all data parallel computations for the problem

d = data parallel computation

$T_{startup}$ = start-up overhead

T_{comm} = per cycle communication cost

T_{comp} = per cycle computation cost

Abstract

A *metasystem* is a shared ensemble of workstations, vector, and parallel machines connected by local- and wide-area networks. The large array of heterogeneous resources in the metasystem offers an opportunity for delivering high performance on a range of applications. Achieving high performance requires effective scheduling of system resources.

This dissertation explores one dimension of the scheduling problem — automatic scheduling of data parallel computations in local-area metasystems containing workstations and multicomputers. Scheduling requires that the problem be decomposed into a set of tasks and data and assigned to processors in a manner that reduces completion time. Problem decomposition is known as partitioning and task assignment is known as placement. Scheduling also requires that the best subset of available processors be selected. No existing system solves all of these problems.

We show that scheduling can be performed automatically, efficiently, and profitably for a range of parallel computations in this environment. A framework has been developed to study the scheduling problem. The framework implements several scheduling heuristics that automate processor selection, partitioning, and placement. At the heart of the framework is a model for representing program and system resource information. From this information, a set of cost functions are constructed to predict computation and communication costs that guide the scheduling process. Scheduling results in a load balanced decomposition of the problem at an appropriate computation granularity.

A framework simulator called Prophesy and a framework implementation in the Legion parallel processing system called Prophet have been completed. The Legion implementation has been applied to a number of real data parallel applications. The results indicate that excellent performance is obtained, scheduling overhead is small, and the costs of heterogeneous parallel processing, format conversion and routing, can be tolerated. A simulation study confirms the performance results and is validated by the experimental results.

Chapter 1 Introduction

Parallel processing in a heterogeneous network environment has become an attractive option for delivering high performance on a range of applications. Interest in distributed parallel processing has been based on advances in three technology areas, local- and wide-area high performance networking [4][6][19][43][81], toolkits that enable network-based parallel processing and job multiprogramming [11][52][73][83], and parallel compilation techniques for distributed-memory MIMD computers [12][33][41][57][76].

In this thesis we consider a distributed computing environment known as a *metasystem*. A metasystem may contain high performance workstations, parallel computers, and vector computers connected by one or more networks, see Figure 1.1. This ensemble of machines presents a large aggregate computing resource including memory, cycles, and communication bandwidth. For this reason a metasystem has a great potential for parallel computing.

An important characteristic of a metasystem is that it exhibits heterogeneity of many types — including hardware, operating system, file system, and network heterogeneity. Heterogeneity poses a challenge in that it must be managed to enable the parts of the metasystem to work together, but it also presents an opportunity — the variety of different resources suggests that it may be possible to select the best resources for a particular problem. The variety and amount of computing resources in the metasystem offers a great

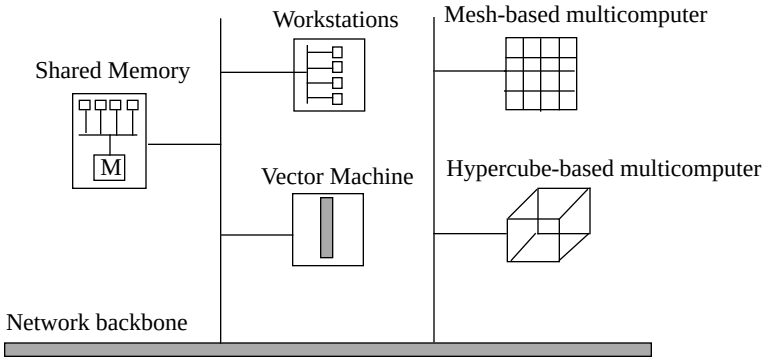


Figure 1.1: A typical metasystem

potential for high performance computing.

Scheduling is critical to realizing the potential for high performance. Scheduling is a difficult problem — the general problem is NP-complete — and effective heuristics that automate scheduling must be used. One of the primary drawbacks of current tools and systems is that they offer limited scheduling support. The programmer is responsible for problem decomposition across the set of heterogeneous processors. This includes partitioning the problem into tasks, selecting processors, and assigning tasks to processors. This tedious and often machine-dependent process has limited the programming of high performance codes to expert programmers in this environment. It is our thesis that scheduling can be performed automatically, efficiently, and profitably for a large class of parallel computations in the heterogeneous environment.

In this thesis we consider one dimension of the scheduling problem — the scheduling of data parallel computations across networks of heterogeneous workstations and multicomputers in a local-area metasystem as depicted in Figure 1.1. Data parallelism is a widely used paradigm for expressing parallel computations and is common to problems in scientific computing. It is an attractive paradigm due largely to the conceptual simplicity of the underlying computational model and the relative ease of implementation. A data parallel model known as SPMD (single-program-multiple-data) has been adopted. The SPMD model been shown to have an efficient implementation in MIMD computers and workstation networks [41].

We deal with two forms of heterogeneity in this metasystem environment, different processor capabilities (e.g., peak Mips and Mflops) and communication capacities (e.g., latency and bandwidth). Scheduling exploits differences in both processor power and communication capacity. We also treat another source of heterogeneity, data format conversion, and show that this overhead can be amortized in many cases.

We assume that the metasystem is a *shared* resource in which computing resources may be committed to other users. This means that resource availability cannot be predicted at compile-time and scheduling must be performed at runtime. It is the shared nature of the metasystem that provides one of its principle benefits — a low-cost computing resource.

We have developed a three stage framework that has been used to study the scheduling problem in heterogeneous environments, see Figure 1.2. The framework automates scheduling with the objective of achieving reduced completion time while keeping runtime scheduling overhead small. Other metrics such as maximizing throughput through the metasystem or minimizing the cost of charged resources[1], are not considered in this thesis. Scheduling is performed *statically* although a dynamic scheduling capability is compatible with the framework. The framework is not tied to current network or computer technology — it will transition to new technologies as they become available. The framework only requires that cost information about a new network or machine technology be provided.

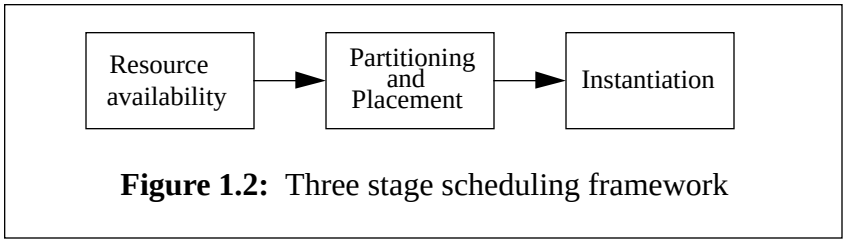


Figure 1.2: Three stage scheduling framework

Resource availability is the first stage of scheduling and determines the state of the available processing resources on the network. Partitioning and placement form the middle stage and are the heart of the scheduling framework. Partitioning divides the problem

1. If some metasystem resources belong to someone else, we may be charged for their use.

into a set of tasks and data, and selects the best processors to use from the available set. Placement assigns tasks to processors. An example of partitioning and placement is given in Figure 1.3 — the problem has been decomposed into four tasks (circles) and four associated data regions (shaded rectangles), and the tasks are assigned to four processors (squares) with one processor not used. Instantiation initiates the data parallel computation using information provided from the middle stage. This thesis deals principally with the middle stage, partitioning and placement. The framework may be implemented within any parallel processing system that can provide a mechanism for determining resource availability and for performing instantiation.

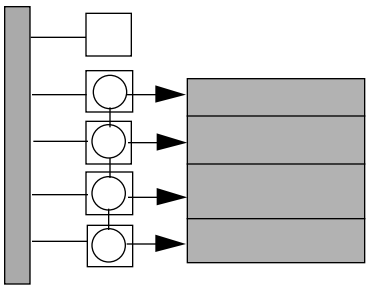


Figure 1.3: Scheduling a data parallel computation

Partitioning is based on achieving an appropriate computation granularity and load balance. An appropriate computation granularity is achieved by selecting processors based on problem characteristics. For example, a small problem will not be able to effectively utilize a large number of processors. This is especially true in a workstation network environment. Constraining parallelism is often needed in this environment due to high communication costs. Load balance is needed to ensure that no processor becomes a bottleneck. This is important in a heterogeneous environment composed of processors with different computational capacities. Placement is based on reducing communication costs. Other metrics for task placement such as memory constraints are the subject of future work. Tasks are assigned to processors using a technique known as *co-scheduling*.

Co-scheduling uses knowledge of the application communication topology and network topology to reduce communication costs such as contention and routing.

Partitioning and placement are guided by cost-based heuristics that use information about the network resources and the computation. Information about the system resources is defined by a heterogeneous network model and information about the data parallel application is defined by a parallel computation model. A set of runtime cost functions that predict the cost of communication and computation are constructed from this information. Using this information, scheduling can make partitioning and placement decisions that are predicted to deliver reduced completion time.

An implementation of the framework in the Mentat-Legion parallel processing system has been completed. Mentat is an object-oriented parallel processing system developed at the University of Virginia [33]. Mentat-Legion is an intermediate form of the Legion system — Mentat is currently being converted to a system (Legion) that will support a wide-area capability. Mentat and Legion are described in the next chapter. The framework implementation is called *Prophet* and has been successively applied to a number of real data parallel codes. Using Prophet we demonstrate that computation granularity, load balance, and co-scheduling are all necessary for achieving reduced completion time and ignoring any one of these can lead to a large increase in execution time. We also show that runtime scheduling overhead is small and the costs of heterogeneity, data format conversion and routing, are tolerable. The performance of the scheduling heuristics have also been confirmed in simulation using the *Prophesy* simulation system. The simulation results indicate that the heuristics have excellent average-case behavior and can be expected to produce execution times within 10% of optimal over 90% of the time.

The organization of this thesis is as follows. Chapter 2 presents related work in scheduling parallel computations, distributed systems, and metasystem computing. Chapter 3 describes the heterogeneous network model, resource availability, and the parallel computation model. Chapter 4 addresses the partitioning and placement problem and pre-

sents two heuristic solutions. Chapter 5 describes the implementation of Prophet in the Mentat-Legion parallel processing system. Chapters 6 and 7 present simulation and experimental results using Prophet. Chapter 8 provides a summary and future work.

Chapter 2 Background

This chapter presents related work in three overlapping areas, scheduling, distributed systems, and metasystem computing. Scheduling is a well-studied topic in and of itself and we present a portion of this vast literature. Distributed systems is also a large and active research area. We present some fundamental results and recent trends in distributed systems research. Finally research in the emerging area of metasystem computing is presented. Work in scheduling and distributed systems has laid the foundation for this new area. We discuss these areas in turn.

2.1 Scheduling

Scheduling is the process of mapping units of work to processors. Research in scheduling parallel computations generally falls into one of two categories — scheduling a directed-acyclic graph (DAG) [1][22][59][97], or scheduling a static-task graph (STG) [9][10][54][80]. The DAG-based precedence graph often arises from the parallelization of sequential code. In the DAG the nodes represent computations, typically fine-grained, and the arcs represent data dependencies. Scheduling an arbitrary precedence graph is NP-complete for $P>2$ processors [87]. Polynomial time algorithms exist for tree-structured DAG's if the nodes are unit time computations and communication is ignored [1], for linear chains [9][65], and when the maximum communication cost is less than the smallest

node computation time and there are sufficient numbers of processors [1]. The DAG encodes temporal information about the computation but may fail to capture the communication structure of the application when it is implemented as a collection of processes.

In the STG the nodes represent modules or tasks, typically coarse-grain, and the arcs represent communications. There are two variants of the STG, the module assignment graph introduced by Stone [80] for non-precedence-constrained sequential programs and the communication graph for parallel computations. Scheduling an arbitrary STG of the first type is NP-complete for $P>4$ processors [80]. Polynomial time algorithms exist for restricted STG's, trees [9], series-parallel graphs [86], and linear chains [9]. The STG captures the communication structure of the application, but loses the temporal information contained in the DAG. Many scientific problems are naturally expressed by the second type of STG — collections of communicating processes with regular precedence and communication relationships. Consequently, the STG is a natural way to express single-program-multiple-data (SPMD) computations. The scheduling model adopted in this thesis is based on a SPMD model of computation. A model that attempts to capture the advantages of both the DAG and STG is the temporal communication graph (TCG) [55] though the efficacy of this model has not yet been demonstrated on real parallel computations.

Scheduling parallel computations has two parts, partitioning and placement. These two parts are often accomplished in several steps. Partitioning determines the schedulable work units and placement assigns these work units to processors. Scheduling is one of the most overloaded terms in the literature. In the distributed systems literature, scheduling is often synonymous with placement only. In the operating systems literature, scheduling is the process of deciding which task will run next. Placement is also called mapping, allocation, assignment, and embedding in the literature.

Scheduling techniques for parallel computations can be classified by the target environment — shared-memory MIMD (SM), distributed-memory MIMD (DM), or dis-

tributed systems (DS). Partitioning and placement are performed differently in these environments.

Scheduling approaches can also be categorized by *when* the scheduling decision is made, compile-time (CT) or runtime (RT), and by whether the decision is static or dynamic. A static scheduling decision does not change while a dynamic scheduling decision may change at runtime. The possible couplings are CT-static, RT-dynamic, and RT-static. The advantage of runtime scheduling is that is possible to consider resource availability and problem information known only at runtime. Dynamic scheduling has the added advantage that it can respond to changes in resource availability and problem workload distribution during the course of execution. The penalty for runtime scheduling is overhead. On the other hand, compile-time scheduling schemes have the advantage of low overhead but often require precise program and resource availability information. We provide a taxonomy of scheduling approaches in Figure 2.1 and discuss them in the subsequent sections. We show only distributed schemes for runtime scheduling. We discuss only a subset of the approaches given in Figure 2.1.

2.1.1 Compile-time Scheduling

Compile-time scheduling is a static scheduling process. Most approaches begin with a labelled graph that must reflect accurate costs for computation and communication. Graph nodes represent computation and arcs represent communication cost. The STG and DAG models define nodes and arcs somewhat differently. Stone presents a STG model where nodes are modules of a sequential non-precedence-constrained program and arcs are module invocations. He extends this graph to represent all possible assignments of modules to processors. A network flow algorithm is then used to solve the module assignment problem for P=2 processors [80]. Much of the research on scheduling STG's is based on this early classic work. Bokhari extends Stone's work to allow module relocation during execution [9]. Modules are executed in one or more phases and it may be advanta-

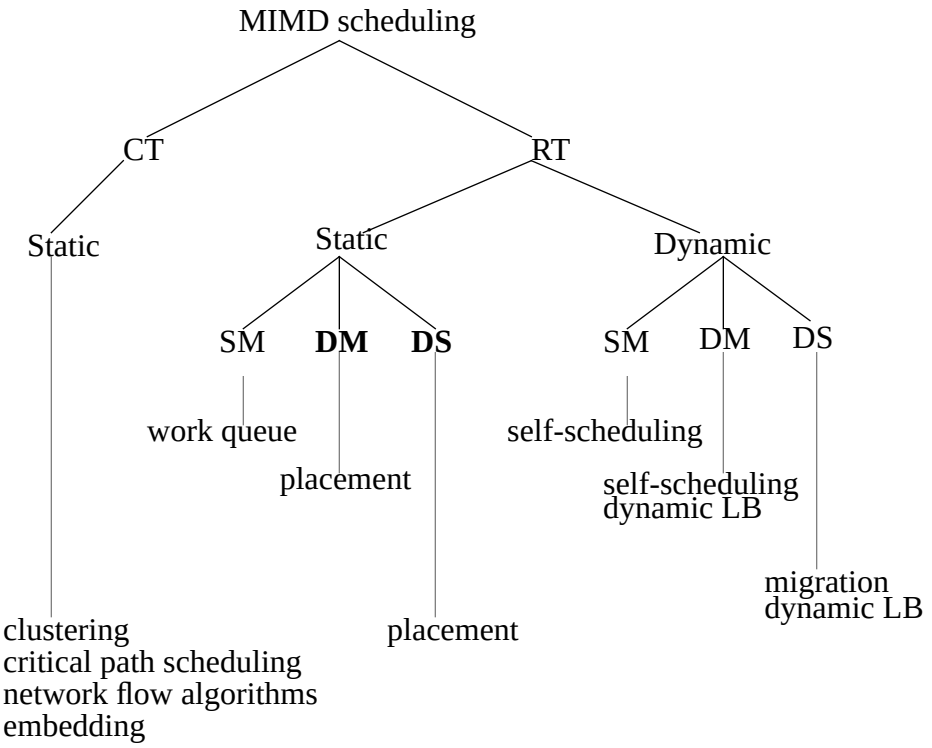


Figure 2.1: Taxonomy of traditional MIMD scheduling techniques. The bold letters indicate where our approach falls.

geous to relocate modules between phases. Bokhari also presents a polynomial time algorithm for tree-structured graphs that is based on Djikstra's well-known shortest path algorithm. This algorithm applies for arbitrary numbers of processors.

An alternate formulation of the STG for parallel programs is a representation of the communication graph [7][54]. Here the nodes are tasks or processes and the arcs represent communication. The problem of assigning such a graph to the processors of a parallel machine has been well studied [48][56][70]. The placement of tasks depends on the communication topology of the graph and the interconnection topology of the parallel machine. Fortunately the topology of many parallel computations falls into a small set of regular topologies. Algorithms for placement have been developed that exploit the topology of the program and the topology of the interconnect. This is sometimes referred to as

graph embedding. The objective is to minimize communication hops and link contention. Our model has been designed to utilize these embeddings.

A great deal of research into compile-time scheduling of precedence-constrained DAG's has followed Stone's initial work [80]. We present a small part of this vast literature. Research has centered on the development of polynomial time algorithms for special cases of the general problem. Bokhari has developed a polynomial time algorithm for linearly-dependent chains on host-satellite systems that contain a time-shared host and a dedicated satellite processor [9]. This algorithm was subsequently improved by Nicol and O'Hallaron [65].

McCreary and Gill have developed a graph clustering technique that takes a fine-grain DAG and produces a coarse-grain graph suitable for execution on a parallel machine [59]. This technique is useful for certain graph structures such as linear chains or series-parallel graphs. Yang and Gerasoulis have developed a scheduling algorithm for coarse-grain DAG's in which scheduling is performed in four phases: clustering, cluster merging, physical mapping, and task ordering [97]. The nodes of the DAG are tasks. Clustering is the mapping of tasks to clusters and attempts to trade-off parallelism and communication overhead. Cluster merging is performed when the number of processors is less than the number of clusters and is done to give load balance. Mapping assigns clusters to processors based on topology and locality. Task ordering within a cluster is done to minimize time on the critical path. It should be mentioned that each of these sub-problems are NP-complete and heuristics are presented.

El-Rewini and Lewis have developed a scheduling algorithm for coarse-grain DAG's [22]. The algorithm is a two phase process: clustering and communication scheduling. Communications are scheduled using topology and routing information. Contention is considered on a link by link basis and is used to avoid high congestion routes. The authors do not consider contention when making clustering decisions. This is probably

because they are more interested in compute-intensive problems in which a precise characterization of communication costs and contention is less important.

2.1.2 Runtime Scheduling

Compile-time scheduling approaches work well when accurate cost information is available statically. It may not possible to obtain accurate static information for computations with data- or control-dependencies or when the processing resources are shared with other users. Runtime scheduling can respond to changes in resource usage and workload characteristics. Information about the computation and the state of processing resources can be exploited by deferring scheduling decisions until runtime. Runtime scheduling has used extensively in distributed systems due to the need to support sharing of processing resources. Runtime scheduling in multiprocessors and multicomputers has also been an active area of current research.

A major difficulty with static scheduling is that it is unable to respond to load imbalance due to problem and system characteristics. Irregular data-dependent computations often have this property. A runtime scheduling technique known as self-scheduling [84] has been developed to address this problem. The basic idea is that instead of a fixed assignment of work to processors, the processors request work from the system when they are finished with their previous task and are idle. The goal of self-scheduling is to try to have the processors finish at the same time. This technique works particularly well for parallel loops with a high execution variance among different iterations. Variations of this technique have been proposed such as tapering [58][69] in which the system adaptively adjusts the size of the work chunk that is assigned based on problem characteristics. If there is little variance in the computation, assigning larger chunks of work is more efficient due to the overhead of work assignment.

Self-scheduling could be viewed as a static runtime technique in that once a processor receives a unit of work or task it is executed to completion. On the other hand, the

method is dynamic in the sense that there is not a single work distribution phase at the outset with predictable work assignments.

While self-scheduling and its variants attempt to avoid load imbalance, dynamic load balancing attempts to detect and then correct the load imbalance. This is a very difficult problem that often arises in data parallel scientific computations [40][50][95]. Many of these scientific applications have the property that the amount of computation performed on a region of the data domain may change unpredictably during the course of execution. Unstructured mesh problems and particle-in-cell simulations are two examples. Dynamic load balancing strategies are used to redistribute the data domain in a manner that attempts to load balance the processors and preserve communication locality.

One of the problems with dynamic load balancing strategies is that the communication costs needed to redistribute data may outweigh the benefits. This is particularly true of centralized as opposed to distributed schemes. The detection of load imbalance may also be expensive since this is often requires some form of global communication. A good survey of techniques is given in [40]. Kumar et al have analyzed the scalability properties of a number of dynamic load balancing schemes on a range of architectures [47]. Near optimal load balancing strategies are presented and analyzed for the hypercube, mesh, and networks of workstations.

Nicol and Reynolds have analyzed the dynamic load balancing problem at a much coarser level [66]. The authors present a decision model for the application of dynamic load balancing for a class of computations. This model is suitable for data parallel computations that exhibit well-defined phase changes. Dynamic load balancing may be required between these phase changes.

2.1.3 Partitioning and Processor Selection

A number of researchers have studied the relationship between problem partitioning and the number of processors that can be used effectively [17][39][64][71]. Gupta pre-

sents a runtime cost-based technique for determining the number of processors to apply to a problem in a shared-memory multiprocessor [39]. Selecting the number of processors to use provides a form of granularity control and determines the problem partitioning. Cytron presents a method for determining the optimal number of processors to use under the simplifying assumption that the communication cost is independent of problem size [17]. Reed et al have studied the impact of data partitioning on the performance of stencil problems [71]. Nicol has analyzed the partitioning problem for stencils to determine the relationship between performance and a number of system parameters including the number of processors. All of this work is based on a multicomputer or multiprocessor environment — a homogeneous environment of dedicated resources. No implemented system in the literature performs the processor selection process automatically. We have developed a processor selection technique that is applicable to heterogeneous networks of shared computers. It has been implemented and applied to real programs.

2.2 Distributed Systems

Much of the research in metasystem computing is based on advances in four related fields of distributed systems research — distributed operating systems, scheduling in distributed systems, toolkits for distributed computing, and parallel processing in distributed systems.

2.2.1 Distributed Operating Systems

An active area of research in distributed operating systems is the accommodation of heterogeneity [8][67][68][98]. Many of these systems deal with heterogeneity of many kinds including processor type and file system differences. Much of this research is concerned with accommodating these heterogeneities in a transparent manner. Few of these systems attempt to exploit heterogeneity since high performance is not a primary goal.

One particular problem in accommodating heterogeneity is of interest in our research, data format conversion. Data format conversion must be performed efficiently if

high performance is to be achieved [96][99]. Conversions are needed for floating point format differences, alignment differences, byte ordering differences, and size differences. The differences may be due to the hardware, operating system, or the compilers used. If formats differ in the range of values that can be represented, it may not be possible to perform a transformation [98].

Data format conversion is handled in one of two ways, either a common format such as XDR [82] is used or application-specific conversions are employed. A common format requires both encoding and decoding of data while the application-specific conversions are one-way only and are much less expensive. Our results indicate that the use of application-specific conversions is about an order of magnitude faster than conversions based on XDR. These results agree with results reported for the Mermaid system, a heterogeneous distributed shared memory system that uses application-specific conversions [99].

2.2.2 Scheduling in Distributed Systems

Scheduling in distributed systems is concerned with achieving an acceptable level of system performance by *load sharing*. Under load sharing job workload is shared among a set of hosts [21]. Jobs will be transferred from heavily loaded to lightly loaded processors. This is a weaker condition than load balance which insures that the processor queue lengths are equal. The most common metric for studying scheduling performance in distributed systems is job throughput. Casavant and Kuhl present a taxonomy of scheduling approaches in distributed systems [14].

Eager and Lazoswka develop a queuing theory model of adaptive load sharing policies for homogeneous systems consisting of a network of computers [21]. The jobs are independent sequential tasks with Poisson arrival that do not communicate and no information about the jobs is otherwise assumed. These load sharing policies consist of a *transfer* policy and a *location* policy. When a processor receives a job for execution, a transfer

policy is used to determine if the job can be scheduled locally. If not, a remote processor is chosen by invoking the location policy. A transfer limit provides stability on the load sharing algorithm. The transfer policy is a simple threshold policy that is based on the queue length and the location policy is a *sender-initiated* scheme. The authors conclude that simple load sharing policies, e.g. location policies that gather a small amount of system state, perform better than no load sharing, and almost as well as more complex policies that will incur larger runtime overhead.

Mirchandaney et al [61] present a queuing theory model for heterogeneous systems that is an extension of [21]. The performance of a simple heterogeneous system consisting of two heterogeneous cluster types was analyzed using threshold policies similar to [21]. A simple sender-initiated policy outperforms a random policy that does not use any information. Some results on choosing the threshold limits are also presented. The conclusion offered by both Mirchandaney et al and Eager et al is that simple scheduling policies perform best. But the results indicate that performance can suffer dramatically under high load. For this reason and others (see Section 2.2.4) load sharing is inappropriate for scheduling data parallel computations that demand a large share of system resources.

2.2.3 Distributed Toolkits

A large number of toolkits have emerged that support scheduling in distributed systems [52][73][98]. In contrast to distributed operating systems, these toolkits are normally layered on top of the existing base operating system and perform a single resource management task, namely scheduling. These systems differ in scalability, load sharing method, and job type supported. Both Utopia [98] and DQS [73] support both sequential and parallel jobs. A parallel job may contain multiple tasks. DQS also supports PVM [83] jobs. Condor [52] attempts to locate idle cycles and is targeted to long-running batch jobs such as simulations.

Condor also favors workstation autonomy — only idle machines can be selected for remote execution, and jobs will be migrated if the selected workstation becomes busy. Utopia, on the other hand, views all system resources as implicitly shared and will not migrate jobs. Utopia also uses application resource requirements and load information to match jobs to processors. Utopia is targeted to heterogeneous networks that may contain thousands of workstations and implements scalable load sharing techniques based on a clustering of processors. Both Utopia and DQS allow resources to be marked private and removed from the shared resource pool. All of these toolkits are limited to workstation networks and are not designed for metasystem environments.

2.2.4 Parallel Processing in Distributed Systems

Many of the assumptions made in scheduling sequential jobs in distributed systems are inappropriate for scheduling parallel computations. Parallel computations consist of a set of related tasks that may communicate during the course of program execution and often require full utilization of the available processing resources. These requirements violate the assumptions of most load sharing algorithms for distributed systems [21]. Furthermore, these algorithms are designed to achieve high job throughput and not necessarily fast completion time for a particular job or task. Data parallel computations, on the other hand, often proceed at the rate of the slowest task and are typically scheduled to minimize completion time.

A number of systems have been developed to support parallel processing in heterogeneous distributed systems. These systems differ in the level of support that is provided. Systems such as PVM [83], P4 [11], and Linda [12] provide the programmer with the basic set of primitives needed for heterogeneous parallel processing but require that the programmer operate at a fairly low-level. In particular, the programmer is responsible for problem decomposition and task placement. PVM is the most widely used system for heterogeneous parallel processing. It provides software to manage a configuration of hetero-

geneous hosts and a library that provides a basic message-passing capability to application programs. PVM supports the notion of process groups and provides several group communication operations, multicast, broadcast, and barriers. PVM also provides a set of data conversion routines for scalar data types to support communication between heterogeneous hosts. PVM provides the necessary building blocks for heterogeneous parallel computing, but the interface is low-level.

P4 supports a wider range of computation models than does PVM — including typed message-passing, shared-memory, and monitors. The support of multiple models makes P4 a larger and more complex system than PVM. P4 does support some higher-level abstractions such as global reduction operations, but it is otherwise a low-level system. Like PVM, the programmer must create and manage processes and use low-level communication routines or shared-memory. P4 uses a common data format, XDR, to perform format conversions in support of heterogeneity. As an optimization, format conversion is performed only when necessary.

Linda provides a higher-level abstraction for communication based on a shared tuple space. The tuple space operates like a shared associative memory — read operations are performed by extracting from the tuple space (*out*) and write operations by inserting into the tuple space (*in*). Since the programmer is aware of the tuple space and must explicitly manage its contents without compiler assistance, we place Linda in the category of low-level systems.

All of these low-level systems provide a basic set of facilities that allow the programmer to execute parallel programs in a heterogeneous environment. There is minimal support for problem and data decomposition — the programmer is responsible for creating and managing processes, communication, and scheduling. While these systems accommodate heterogeneity to some extent, they do not exploit heterogeneity.

A number of systems that provide greater support for heterogeneous parallel processing have emerged over the past few years [27][31][33][62][76]. These systems may be

distinguished by the level of compiler and runtime system support for managing parallelism and scheduling. Mentat [33] is an object-oriented parallel processing system. Mentat programs are written in MPL, a high-level language based on C++. The programmer specifies the grains of computation by indicating that a class is a *Mentat class*. A Mentat class contains member functions of sufficient computational weight to warrant parallel execution of Mentat class instances. Instances of Mentat classes, known as Mentat objects, are implemented by address-space disjoint processes, and communicate via methods. Method invocation is accomplished via an RPC-like mechanism. A strategy for supporting data conversion of arbitrary data types is discussed in [31]. Mentat also performs runtime scheduling [37] based on Eager and Lazoswka's adaptive load sharing [21]. Support for scheduling data parallel computations in heterogeneous environments has been recently added to the runtime scheduler [31][91] as part of this thesis.

Data parallelism is expressed in Mentat by defining a Mentat class that corresponds to a SPMD task and instantiating some number of Mentat objects of this class. In Mentat, the programmer is responsible for choosing the number of Mentat objects and decomposing the data domain. Mentat does automate the placement of Mentat objects to processors but does not use any program information to do so. Scheduling in Mentat is based on Eager and Lazowska's adaptive load sharing model [21].

Charm [76][77] is an object-based parallel processing system based on a message-driven execution model. The grains of computation are specified by the programmer using a language construct called a *chare*. Chares resemble Mentat objects to a certain extent — they encapsulate data, they have a well-defined typed interface that specifies the allowable operations, and their operations are executed in a monitor-like fashion. Charm also provides runtime scheduling for chares. Chares are scheduled using an adaptive load sharing algorithm that is based on the load of the processors that fall within a local neighborhood. In Charm processors periodically exchange load information with the set of processors in this neighborhood.

Dataparallel C [41][62] is a high-level language and runtime system that supports programming data parallel applications. Dataparallel C programs are written in a shared-memory style using data parallel constructs. The compiler and runtime system handle program and data decomposition. In Dataparallel C the basic unit of work is the virtual processor and virtual processors are assigned to physical processors. The virtual processor can be thought of as a basic unit of the data domain. The scheduling support is limited — the programmer specifies how many processors to use. The runtime environment is targeted to heterogeneous workstations and a dynamic load balancing strategy is provided.

A number of systems provide explicit support for scheduling data parallel computations on a network of heterogeneous workstations [5][13][16][33][62][76][78]. The Dataparallel C runtime system implements a dynamic load balancing strategy for regular, iterative data parallel computations. Each processor participates in a four stage dynamic load balancing algorithm, load screening, exchange of load information, migration decision, and migration action. Load screening is accomplished by inserting timers around the virtual processor execution code. The processor load is the average computation time per virtual processor — this is known as the load index. This measure assumes that the amount of computation per virtual processor is the same throughout the problem. The time between successive load information exchanges is set to be a small fraction of the average time taken to do a migration. Migrations consist of moving virtual processors from processors with a high load index to processors with a smaller load index. Processors are not free to migrate data to any processor since locality relationships in the problem domain must be maintained. Dataparallel C is not applicable to the metasystem environment and is suitable for regular parallel computations only. The system is further limited by the assumption that the programmer specifies the number of processors to use.

Charm [76] solves a simpler dynamic load balancing problem than Dataparallel C. In Charm tasks are assumed to be labelled with a task finishing time so a processor can determine how much work the task has remaining. Tasks can be freely moved to any pro-

cessor — this scheme will only work for problems that do not have communication locality. One weakness is that the cost of migration is not considered.

The Paragon project [16] addresses the problem of static partitioning a data parallel computation on a network of heterogeneous workstations. The Paragon system determines a load balanced decomposition and addresses the problem of choosing the number of processors to use. The approach is based on benchmarking a number of common parallel operations on all possible configurations of a heterogeneous network. This information is used to form a performance prediction for a given code and a table-driven method for choosing the best configuration of processors has been implemented. Most codes in Paragon will be constructed as combinations of these common parallel operations. Their solution will not scale to large numbers of processors in which benchmarking all possible processor configurations is not feasible. Our approach requires a much simpler benchmarking strategy in which the sequential code is benchmarked once on each machine type.

Attalah et al [5] have also studied the problem of processor selection on a network of heterogeneous workstations. This work is targeted to compute-intensive data parallel computations. The authors present a model of the processor's capacity called the *duty cycle*. The duty cycle is a load index that is defined as the ratio of cycles committed to local, non-compute-intensive tasks to the number cycles available for compute-intensive tasks. Only a single compute-intensive task will be scheduled on a processor at a time. If a processor is already running a compute-intensive task, it is removed from the current pool of available processors. Use of this processor for a new scheduling request will delay the time at which this computation may begin. This is known as *gang scheduling* — the computation will not begin until all selected processors are ready (i.e., have no currently running compute-intensive tasks). The scheduling algorithm tries to minimize the sum of the weighting time and the expected computation time. This approach is limited by the assumption that communication costs are negligible.

Piranha [13] is an extension of Linda that supports a scheduling concept known as adaptive parallelism. In adaptive parallelism the number of processors applied to a computation may shrink or grow during the course of execution. Processors will not be allowed to leave if they are currently executing a task. Piranha is a master-worker model that is based on Linda's shared tuple space. One major problem with this approach is that the master will become a bottleneck for large systems and this limits the scalability of this approach.

2.3 Metasystem Computing

Metasystem computing is a natural progression from the research in parallel processing and distributed systems. Many of the issues inherent in metasystem computing are described in [27][45][46]. These issues include code matching, scheduling, programming environments, and performance evaluation. Code matching defines an affinity between a schedulable program component and a machine type. A class of programs suitable for metasystem computing contain several large-grain code modules that may exhibit different types of embedded parallelism or affinities. The benefit of exploiting program affinities for specific applications has been demonstrated by a number of research groups including [24][60][63]. In a global climate model code [60], decomposing two large-grained program components across a Cray Y-MP and an Intel Paragon resulted in superlinear speedup with respect to running the program entirely on the Y-MP or the Intel Paragon. The program component assigned to the Y-MP was highly vectorizeable and the component assigned to the Paragon was data parallel. Other researchers have reported superlinear speedup and the conditions for achieving superlinear speedup are discussed in [20].

Many of the metasystem applications contain program modules that have been optimized for particular hardware and a great deal of effort goes into glueing the program modules together. These applications must manage the complex details of integration and

heterogeneity as part of the code. A number of software systems for metasystem computing have emerged [34][42] to help facilitate program integration and metasystem execution. Schooner promotes integration by providing glue software that supports RPC, a module description language for specifying and connecting modules, and a common data format. Schooner is geared toward the integration of loosely-coupled modules and does not have high performance as a stated goal. For example, the use of a common data format adds significant overhead for tightly-coupled parallel computations.

Legion is a software framework that promotes integration but not at the expense of high performance [34]. The high performance objectives of Legion have been inherited from the Mentat project [31][33]. Legion supports efficient parallel and distributed computing by adopting the Mentat model of computation and by providing runtime scheduling support [32][36]. The goal of the Legion project is to provide a seamless virtual machine that may contain computers connected by LANs, MANs, and WANs. The goal of efficient wide-area computing separates Legion from most other contemporary systems.

A number of research groups are exploring a concept known as *superconcurrency* or *heterogeneous supercomputing* [15][18][23][26][27][45][88]. An important distinction between this body of work and other efforts in parallel processing in heterogeneous networks is that superconcurrency is concerned with choosing the best subset of available machines as opposed to load balancing. Machines are also assumed to be non-shared. In the superconcurrency model, programs contain a number of large-grain modules called code segments, and code segments contain a number of code blocks. Code segments are assumed to be executed in a sequential fashion. There may be parallelism between code blocks. The approach is based on two techniques developed by Freund [26], code profiling, and analytic benchmarking.

Code profiling determines what types of code blocks or segments a program contains. Code types include vectorizeable, decomposable, SIMD, or MIMD. Analytic benchmarking determines how well codes of a given type are expected to perform on the

different machine types. These techniques are not described in sufficient detail in the superconcurrency literature. Freund also defines the assignment of code blocks or segments to machines as a mathematical optimization problem that minimizes completion time subject to a cost constraint. This is a compile-time mapping problem and assumes an unlimited supply of dedicated machines. Another limiting assumption is that communication between code segments is ignored.

The Augmented Optimal Selection Theory (AOST) extends Freund's work in two ways [15] — a finite number of machines is assumed, and a more accurate cost model for code types is developed. Code profiling is used to produce an affinity value for each code block/machine type pair. In Freund's approach only the affinity for the optimal machine type was benchmarked. The affinity for a non-optimal machine type was estimated to be a scalar speedup value. The authors point out that this can lead to an underestimation of the affinity for a non-optimal machine type. AOST also allows different machine models in the same machine class. For example in a hypercube machine class, the iPSC/2 and iPSC/860 would be treated differently. A decision algorithm for compile-time machine selection is provided. This model also assumes no parallelism between code segments.

Several superconcurrency projects have relaxed the restriction of no parallelism between code segments [15][23][44]. The Heterogeneous Optimal Selection Theory (HOST) extends AOST to allow parallelism between code segments. This approach is based on a programming paradigm known as Cluster-M [23]. Cluster-M is a graph-based language for expressing task decomposition, code types, communication relationships, and parallelism opportunities between code segments. Cluster-M is also used to graphically represent the available machines in a hierarchical fashion. This paradigm exposes the communication topology and interconnection topology and is exploited by a mapping heuristic. The authors claim that this technique can be used for finer-grain computations. No results are reported for this heuristic. Iqbal [44] presents an optimal scheduling proce-

dure for mapping a linear chain of code segments onto an array of heterogeneous computers.

All of these efforts are based on a static, compile-time assignment of program modules to a set of dedicated heterogenous machines. Dietz et al have developed an approach called Augmented Heterogeneous Selection (AHS) which relaxes the assumption that the machines are dedicated. Two parallel specification languages, MIMDC and SIMDC, are provided to allow users to express parallel computations. The execution cost of the program is determined at compile-time by summing up the component costs. The cost of computation and communication is determined for each machine by off-line benchmarking. This cost estimate is adjusted at runtime to reflect current processor load. The load adjustment as well as the estimate of computation and communication cost does not consider a number of factors including memory costs and communication contention. But unlike the earlier work in superconcurrency, they are not interested in optimal results, but in a practical system that can be shown to deliver good performance.

Most of the applications developed for metasystem computing environments contain large-grain heterogeneity. A number of researchers are looking at finer-grain problem heterogeneity and have proposed reconfigurable hardware designs to support these types of applications [2][51][89]. Watson et al introduce a SIMD/SPMD mixed-mode machine designed for applications that contains SIMD computations coupled with SPMD computations. These applications typically cycle between SIMD and SPMD computations and the hardware dynamically adjusts to the proper computation mode. Ligon and Ramanchandran propose a reconfigurable architecture known as a multigauge architecture. The multigauge architecture configurations are limited to bit-serial SIMD modes. It has been successively applied to image understanding problems such as the DARPA image understanding benchmark [90].

Chapter 3 The Models

This chapter presents the heterogeneous metasystem model and the parallel computation model. These models lay the groundwork for the scheduling framework discussed in the next chapter. The metasystem model provides a representation and organization of system resources and defines the important resource information needed by the scheduling framework. This information is used in two ways — to determine resource availability and to construct cost functions for computation and communication. These cost functions are needed to support scheduling. In particular, a set of off-line communication functions provide an accurate estimate of expected communication costs and are used in the processor selection process. Similarly, the parallel computation model provides a representation for parallel programs and defines the program information also needed by the scheduling framework. Program information is used to select the appropriate communication cost function based on the application communication topology, to construct the computation cost function based on the problem characteristics, and to provide parameters to the cost functions such as message size.

3.1 Metasystem Model

The metasystem model has two parts, the network organization, and the communication model. We present a scalable network organization for representing both local- and

wide-area resources. We also present a communication model that is used to determine the cost of communication between machines in the metasystem.

3.1.1 Network Organization

The basis of the network organization is the *processor cluster*. A processor cluster contains a homogeneous family of processors that may include workstations, vector, or parallel machines. A vector machine would be treated as a uniprocessor, i.e., a cluster containing one processor. A parallel machine would be treated as a single cluster of processors. The processors in a processor cluster share communication bandwidth. Processor clusters may range from tightly-coupled multiprocessors such as a Sequent in which processors communicate via shared-memory, to distributed-memory multicomputers such as a Paragon or loosely-coupled workstations such as a Sun 4 cluster in which processors communicate via message-passing. This particular configuration is depicted in Figure 3.1. The processor clusters are denoted by the large circles. Each processor cluster has a *manager* denoted by the shaded circle. For multicomputer-based processor clusters the manager would be an external host processor. The role of the manager will be discussed shortly.

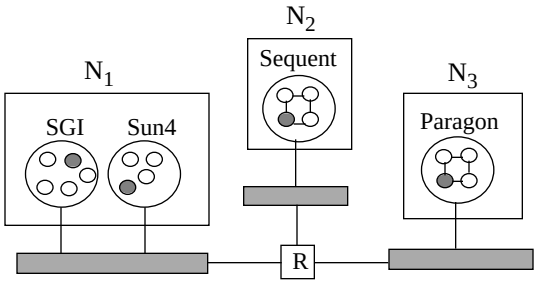


Figure 3.1: Cluster-based metasystem organization

A *network cluster* contains one or more processor clusters and is denoted by the boxes labelled N_1, N_2 and N_3 in Figure 3.1. The essential property of a network cluster is that it has private communication bandwidth with respect to other network clusters, and shared bandwidth with respect to the processor clusters it contains. For example, the total

available bandwidth in the metasystem of Figure 3.1 is the sum of the bandwidth in N_1, N_2 and N_3, but the available bandwidth in N_1 is shared between the Sun 4 and SGI processor clusters. Each network cluster has a *network cluster manager.* Network clusters are connected by one or more *routers*. We use the term router to refer any type of network connector such as a router, gateway, or bridge. The router introduces delay and adds communication cost.

Communication between processors in different processor clusters is accomplished by message-passing. For simplicity we will assume that all communication is by message-passing. This simplifies the presentation of the communication cost functions in the next section. In shared-memory multiprocessors message-passing can be easily implemented on top of shared-memory. Taken as a whole, the metasystem is a multi-level distributed-memory MIMD machine.

We will use the following notation throughout this and subsequent sections[1]:

N_i =	the i^{th} network cluster
C_i =	the i^{th} processor cluster
P_i =	number of processors selected for C_i
P_T =	total number of processors selected
τ =	application communication topology
b =	message size in bytes
c_1 .. c_4 =	communication cost constants
f () =	cluster-dependent communication function
F() =	topology-dependent total communication function
r_1, r_2 =	router cost constants
e_1 =	conversion cost constant
v =	number of messages that cross between each processor cluster

The managers maintain important information about the network resources, see Figure 3.2.[2] The topology refers to the type of interconnect. Examples include *bus* (ether-

1. We have not yet defined all terms, but they will be defined before their use.
2. Not all of this information is used in the current implementation.

net), *ring* (FDDI), *mesh* (multicomputer), and *hypercube* (multicomputer). The bandwidth refers only to network clusters. The *peak* bandwidth is the maximum communication bandwidth achievable for this network cluster assuming idle machines and network (e.g., 10 Mb/sec for an ethernet-based network cluster). The *avail* bandwidth is the amount of the peak bandwidth available based on the current network usage. Latency is the end-to-end cost of sending a 0 byte message between two machines within a processor cluster. Because latency is primarily a processor cost it is associated with the processor cluster. The machine type includes *workstation*, *multicomputer*, *multiprocessor* and *vector* and is associated with the processor cluster.

- Interconnection topology
- Bandwidth (peak, avail)
- Latency
- Machine type
- Communication functions
- Processors (total, avail)
- Memory (real, virtual)
- Aggregate power (mflops, mips)
- Manager

Figure 3.2: Cluster-based resource information

The communication functions provide an accurate measure of the expected communication cost between machines in a processor or network cluster. The latency and bandwidth values can be used to estimate communication costs if these communication functions are left unspecified. Using these latency and bandwidth values provides an optimistic communication cost estimate since contention is ignored. On the other hand, the communication cost functions include contention and application/interconnection topology.

The total processors is the number of physical processors that are contained in a processor or network cluster. The number of processors in a network cluster is the sum of the processors in each contained processor cluster. The available processors are a subset of

the total processors. Processors become unavailable in two ways — they become reserved by other users or the amount of available processing resources on a processor is too little to be considered useful. Memory is the amount of real and virtual memory available within the cluster.

Aggregate power is the cumulative processing power based on the peak instruction rate for the processor type and the number of available processors. The amount of effective cumulative processing power is guaranteed never to exceed this value. For example the amount of Mips or Mflops that a computation actually utilizes depends on the computation. We will see later that a more accurate problem-dependent measure of the effective processing power is made available to the system. If such a measure is left unspecified then the peak rates can be used as an estimate. The aggregate power for a network cluster is the sum of the aggregate power in each contained processor cluster. Some of this resource information must be adjusted to reflect current resource usage. This is discussed in Section 3.1.3.

The manager refers to the name of the processor that stores and maintains the information in Figure 3.2. A manager is associated with each processor cluster and network cluster. One of the processor cluster managers is designated as the network cluster manager. Managers maintain static information such as peak processing power and total number of processors. The information in Figure 3.2 is kept in a resource or configuration database along with a set of cost functions for communication, routing, and conversion described in Section 3.1.2. Managers also monitor and maintain dynamic information such as the available processors. All of this information must be up to date when a scheduling request is made.

In this dissertation we have studied local-area metasystems such as in Figure 3.1 that contain multicomputers and workstations. We make the simplifying assumption of one processor cluster per network cluster. This assumption allows us to present a simpler communication and scheduling model and only limits workstation clusters since by defini-

tion a network cluster can contain only a single multicomputer, multiprocessor, or vector processor cluster. We now discuss several alternatives for wide-area organizations although their implementation is the subject of future work.

Wide-area

A wide-area organization can be defined as a natural extension of the local-area model of Figure 3.1. For wider-area metasystens, we define network clusters hierarchically as shown in Figure 3.3. For example N_4 is a network cluster that contains N_1, N_2 and

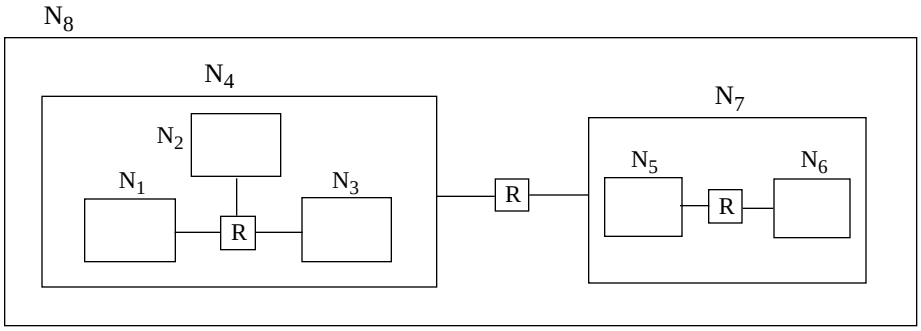


Figure 3.3: Wider-area metasystem organization

N_3. The hierarchical organization of Figure 3.3 forms a tree as shown in Figure 3.4 and captures important communication relationships. The leaves are the processor clusters and communication between processors in a processor cluster (e.g., C_1) does not incur any routing penalty. If processors are in different processor clusters but in the same network cluster (e.g., N_1), the cost is higher due to the single hop routing penalty. Each level of the tree introduces an additional routing penalty.

The network cluster manager stores the names of the managers of contained processor or network clusters to enable exchange of system information. The manager of a network cluster stores an aggregate of the information associated with the network or processor clusters it contains. For example, the total number of processors stored with the manager of N_4 is the sum of the total number of processors of N_1, N_2, and N_3. The same is true for communication bandwidth and aggregate power. The manager stores a copy of the

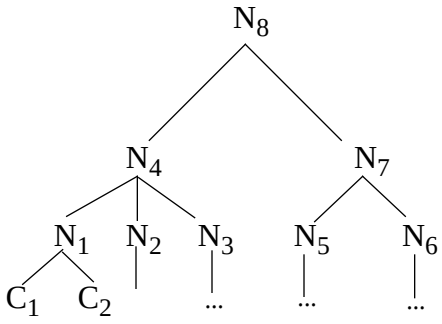


Figure 3.4: Hierarchical metasystem organization

information that is stored with its contained processor or network clusters. For very large metasystems copies of this information can be kept on disk.

It is possible that a network cluster may participate in one or more configurations. For example the user or system administrator may want to define a configuration that contains only N_1 and N_2 and a different configuration that contains N_1 and N_3. Also note that a configuration may be confined to contain a subset of available clusters. Both of these capabilities should be supported in an implementation of the model.

It is unlikely that propagated state information can be kept up to date in the tree organization. By the time information from the leaves reaches the root in a large metasystem it will be stale. A tree also does not exhibit a high degree of fault tolerance. Instead we propose a more scalable and fault-tolerant organization that is based on the concept of *sites*. Instead of a tree at every level, we might organize clusters within a *site* as a tree, and the sites themselves in a completely connected graph, see Figure 3.5 (the circles represent network clusters as in Figure 3.4). Within each site, we would designate the root network cluster manager to be the *site manager* (shaded node). All site managers know each other's identity. A site is an organizational entity that contains network clusters. Examples include universities or government labs. The idea is that only sites would need to maintain up to date state information and the information would not be propagated between sites. The disadvantage of this organization is that less global information is available.

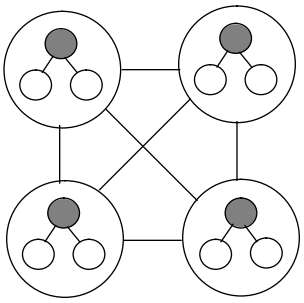


Figure 3.5: Site-based metasystem organization

Resource-based

For wider-area networks it may also be important to expose resource types and make more global information available. For example a program that contains two loosely-coupled data parallel computations might be best served by two Intel Paragons even if they are located in different sites. Another example might be a highly vectorizeable program that would be best served by a single Cray Y-MP that is located remotely. Another possibility is a resource requirement — the computation must run on a set of machine types. Locating a site that contains these machines may be difficult due to the absence of global information.

One possibility is to designate a number of site managers as *resource managers*. Resources managers maintain a table that contains an entry for each *machine type* and a list of site managers that manage clusters containing machines of that type. Every site stores the name of the nearest resource manager. Within this table the resource managers would have to be stored in a manner that attempts to retain some locality information. For example a selection of two Intel Paragons connected by a high-speed link may be preferable to two Intel Paragons that are connected by multiple, slower links. A resource-based organization is most useful for wide-area configurations and programs with resource affinities.

We speculate that the site-based organization with some mechanism for exposing resource information is likely to be an effective model. Future research is needed to confirm this conjecture.

3.1.2 Communication Model

Estimating the communication cost between machines in the metasystem is a central part of the partitioning and placement process. Selecting the appropriate number of processors to apply to a problem depends on the communication cost. For example, choosing too many processors results in high communication costs and increased completion time. Partitioning uses a set of communication cost functions to estimate communication costs for candidate processor selections. An accurate estimate of communication cost will allow processor selection to determine the appropriate number and type of processors to use. These cost functions are based on a message-passing model. We have developed a model that accurately characterizes communication cost for the type of communications that are commonly found in data parallel programs. This cost model also includes two related costs inherent in heterogeneous metasystem communications, routing and data conversion. These cost functions are constructed by off-line benchmarking and are stored by each cluster manager for use at runtime. We begin by discussing the routing and conversion cost functions since they are a part of the general communication cost function discussed in the subsequent section.

3.1.2.1 Routing and Data Conversion

When a message crosses from a processor in one cluster to a processor in another cluster it must cross a router or gateway. This introduces delay due to buffering and routing control. We define the routing cost from a processor in cluster C_i to a processor in cluster C_j to be:

$$T_{router}\,[C_i, C_j]\,(b) = r_1 + r_2\,b \qquad \text{(Eq.3.1)}$$

and by symmetry,

$$T_{router}\,[C_i, C_j] = T_{router}\,[C_j, C_i]$$

We use the square-brace notation to indicate that there is a different function for each parameter value (in the braces) and the parenthesis to indicate the function parameters that are passed at runtime. For example there is a different router function for each pair of clusters and each router function depends on the message-size *b* passed as a runtime parameter. The router cost includes a latency penalty r_1 and a per-byte penalty r_2 that captures any delay or buffering required in routing a *b* byte message from a processor in C_i to a processor in C_j. This cost function is constructed by benchmarking and should be viewed as a lower bound on the actual cost, since routers and gateways are highly shared resources and can introduce unpredictable delays at peak times during the day. A highly loaded router can drop packets and introduce high delays. We model the routing cost from C_i to C_j by a single function even though the communication between C_i and C_j might actually cross several routers or gateways depending on the network configuration. A more complicated alternative would be to model the cost of each router *hop* from C_i to C_j and form the sum. This strategy would make benchmarking routing costs much more tedious.

One way to handle the non-determinism of routing overhead is to provide a set of time-dependent routing functions $T_{router}[C_i, C_j, t]$ which gives the average routing cost at time *t*. At peak times during the day, the routing cost will be higher than at off-peak times. A simpler strategy is to form $T_{router}[C_i, C_j]$ as the average obtained over some large time interval that includes both peak and off-peak benchmarking.

Data format conversion may also be needed for messages that cross between clusters. Conversion is the price paid for using heterogeneous processors. Since processor clusters are homogeneous there is no need for conversion of messages communicated within a processor cluster. Conversion is needed when communicating processors in different clusters support different data formats. Some common conversions include floating point format, alignment, byte ordering, and size [99]. We have studied the most common form of conversion, endian byte re-ordering, and determined this cost by benchmarking. Conversion is paid as a per-byte processor cost by the sending or receiving processor. We

define the conversion cost for a b byte message communicated between C_i to C_j for a conversion of type *conv_type* to be (where e_i is the per-byte cost of a processor in C_i performing the conversion):

$$T_{conversion}[conv\_type, C_i, C_j](b) = e_1 b \text{ with} \qquad \text{(Eq.3.2)}$$
$$T_{conversion}[conv\_type, C_i, C_i] = 0$$

We will drop the *conv_type* in the remainder of the dissertation as we have limited our study to endian conversion only.

In our experience conversion can be easily tolerated even for tightly-coupled parallel computations, if performed carefully. For example consider a simple broadcast topology in Figure 3.6 and suppose the master and workers require format conversion. If conversions are performed by the workers in parallel, the conversion overhead is more easily tolerated. On the other hand, if the master performed the conversions they would be serialized. The placement of conversions can greatly reduce the cost penalty that the application experiences. Another possibility is to assign conversions to the processors that can perform them most efficiently. In the current implementation, conversions are performed by the fastest clusters and are assumed to be performed in parallel as in Figure 3.6. The router and conversion cost functions will be a component of the communication cost described in the next section.

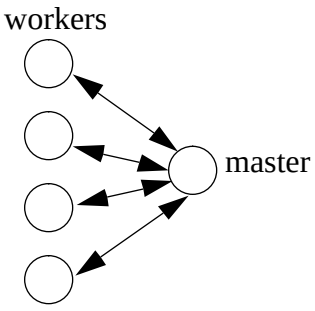


Figure 3.6: Broadcast topology

3.1.2.1 Communication Cost Functions

Scheduling must consider the cost of communication in making partitioning and placement decisions. Effective scheduling requires an accurate characterization of this cost. Consider the simple case where all communication occurs within a cluster C_i (i.e., only processors within C_i are used). The communication cost function for C_i depends on the application communication topology and the interconnection topology of C_i. The particular cost experienced by an application depends on two *application-dependent* parameters provided to this function: (1) the message size, and (2) the number of communicating processors or tasks. There is a one-to-one relationship between tasks and processors in our model — a single task is assigned to a processor. Throughout the dissertation we will refer to communicating tasks and communicating processors, but these terms are synonymous.

The communication patterns for data parallel computations are often regular and *synchronous*. In a synchronous communication all processors participate in the communication *collectively* at the same *logical* time. Scheduling exploits both of these properties. Placement exploits regularity in the communication pattern and partitioning exploits the synchronous nature of the communication.

Our communication model is based on regular and synchronous communications that are performed repeatedly or iteratively during the computation. Although communications are logically synchronous they are asynchronous in the implementation. The synchronous nature of the communication means that the average cost experienced by all processors per iteration is roughly the same and is determined by the processor experiencing the greatest cost. This observation has been verified by empirical data. We demonstrate the generality of our communication model by representing four communication topologies often found in data parallel computations: *1-D*, *ring*, *tree*, and *broadcast*.

The *1-D* is common in scientific computing problems based on grids or matrices and is a class of nearest-neighbor topologies. In the *1-D* topology processors simultaneously send to their north and south neighbors and then receive from their north and

south neighbors. The *ring* topology is common to systolic algorithms and pipeline computations. In the *ring* topology communication is much more synchronous. A processor receives from its left neighbor and then sends to its right neighbor.

The *tree* topology is used for global operations such as reductions. In the fan-in, fan-out *tree* topology communication occurs in two phases. In fan-in a parent processor receives from all of its children before sending to its parent, while children simultaneously send to their parent. Once the root receives from its children the process is repeated in reverse during fan-out.

The *broadcast* is a master-slave topology in which slaves simultaneously communicate with the master, and then wait to receive from the master. A *broadcast* is a global communication that is a special case of the *tree* topology.

A set of accurate communication cost functions can be constructed for each cluster by benchmarking a set of *topology-specific* communication programs. These cost functions determine the average communication cost, measured as elapsed time, incurred by a processor during a single *communication cycle*. A communication cycle corresponds to a single iteration of the computation. For example in a single cycle of a ring communication, a processor receives one message from its left neighbor and sends one message to its right neighbor. For each cluster C_i and communication topology τ, we have a communication cost function of the form: $T_{comm}\ [C_i, \tau]\ (b, p)$.

The cost function is parameterized by p, the number of communicating processors within the cluster, and b, the number of bytes per message on average. For example suppose C_1 refers to the SGI cluster in Figure 3.1. The cost function $T_{comm}\ [C_1, \textit{1-D}]\ (b, p)$ refers to the average cost of sending and receiving a b byte message in a *1-D* communication topology of p SGI processors computed as elapsed time. This cost contains processor and network costs. Processor costs include operating system, protocol, and context-switching overhead. All of these may be quite large for communications on ethernet-connected clusters. Network costs include time spent in the interconnection network. Multi-

computer and multiprocessor communications often incur a much smaller processor and network cost. The communication cost functions have a latency term that depends on p and a bandwidth term that depends on both p and b (c_1 and c_2 are latency constants and c_3 and c_4 are bandwidth constants):

$$T_{comm}\,[C_i, \tau]\,(b, p) = c_1 + c_2\,f(p) + b(c_3 + c_4\,f(p)) \qquad \text{(Eq.3.3)}$$

The first two terms are the latency cost and the later two terms are the bandwidth or per-byte costs. The latency and bandwidth terms both have a component that is independent of the number of processors (i.e., c_1 and c_3) — this would include processor costs such as protocol stack overhead. Each term also has a component that depends on the number of processors (i.e., c_2 and c_4) — this captures contention effects. The function f depends on the cluster interconnect and the application communication topology. For example, on ethernet we often see f linear in p for all communication topologies due to contention for the single ethernet channel. On the other hand, richer communication topologies such as meshes and hypercubes have greater communication bandwidth that scales more easily with the number of processors. For example, we have observed that for tree communication on a mesh, f is *logarithmic* in p. For a *2-D* communication on a mesh f is nearly constant and independent of p since there is limited link contention. Each communication cost function is benchmarked using different p and b values to derive the appropriate constants.[3] The form of this equation has been validated by experimental data.

The communication cost functions depend on the communication system that will be used. For example, on a network of workstations, communication using PVM [83], P4 [11], or raw TCP/IP will have different costs. A different set of cost functions would be needed for these different communication systems. We use a communication library called MMPS (Modular Message-Passing System) [38] which is used by the Mentat-Legion parallel processing system [33]. MMPS is a reliable heterogeneous message-passing system

3. These cost functions are easily generalized for multiple processor clusters per network cluster.

that uses UDP datagrams for communication among workstations and between processors in different clusters, and NX for communication among processors in Intel multicomputer clusters.

A suite of MMPS communication programs has been developed to perform the benchmarking needed to derive the constants in (Eq.3.3). In these programs a set of communicating tasks is assigned to processors. Benchmarking has been done when the processors and network were lightly loaded. The placement of tasks depends on the communication and interconnection topologies and is discussed in Chapter 4. The function in (Eq.3.3) is much more accurate than the often-used communication cost function:

$$T_{comm} = T_{latency} + bT_b \qquad \text{(Eq.3.4)}$$

This communication cost function is normally constructed from two communicating processors and is therefore optimistic — it does not account for contention, topology, or placement. This function provides a lower bound on the expected communication cost. In the event that a communication cost function is left unspecified or unknown, the implementation must construct an approximate cost function based on available information. This is discussed in Chapter 5. If minimal information is available then the cost function of (Eq.3.4) may be used[4].

If the candidate processors considered by scheduling occur within a particular C_i only, then the cost function in (Eq.3.3) determines the communication cost. If processors in several clusters are considered, then communication will cross cluster boundaries and two additional costs may come into play, T_{router} and $T_{conversion.}$ Suppose that processors in C_i are communicating with processors in k different clusters and v_k messages cross between C_i and each cluster C_k every communication cycle. The communication cost for processors in C_i becomes the sum of the previous cost equation in (Eq.3.3) plus several new terms:

4. This function will have to be adjusted to account for contention.

$$T_{comm}[C_i, \tau] = T_{comm}[C_i, \tau](b, p + k) + \sum_{C_k} v_k (T_{router}[C_i, C_k] + T_{conversion}[C_i, C_k]) \quad \text{(Eq.3.5)}$$

Notice that each message sent between C_i and C_k pays a routing penalty and potentially a conversion penalty. It is therefore important to reduce v_k. This is the job of placement discussed in Chapter 4. The experimental evidence indicates that reducing the number of messages to cross the router can significantly lower communication costs. Since the router shares the communication channel we have observed that it increases contention as though the number of processors is increased. This is modelled as k additional stations for k clusters, hence the parameter $p + k$ for T_{comm}. The value of k and v_k depend on the interconnection and application topologies and the placement strategies used.

As an example suppose that processors in C_i and C_j are communicating in a *1-D* topology ($k = 1$). Placement will arrange the communicating tasks such that $v_k = 1$. The communication cost for processors in C_i becomes (C_j may be written similarly):

$$T_{comm}[C_i, \tau] = T_{comm}[C_i, \tau](b, p+1) + (T_{router}[C_i, C_j] + T_{conversion}[C_i, C_j])$$

The cost equation in (Eq.3.5) gives the communication cost experienced by all processors in a particular cluster. The total communication cost experienced by the application depends on the application communication topology and is denoted by $T_{comm}[\tau]$. The total cost is a function F of the individual cluster communication costs:

$$T_{comm}[\tau] = F\{T_{comm}[C_i, \textit{1-D}], \text{ for all selected } C_i\} \quad \text{(Eq.3.6)}$$

We have identified two classes of communication topologies that determine the form for *F, concurrent access topologies* (CAT) and *sequential access topologies* (SAT). These categories are similar to Cytron's concurrent and sequential access paradigms [17]. In a CAT topology processors concurrently send messages asynchronously and then block on message receipt. In a SAT topology processors block waiting for a message and then send a message. In a CAT the communication channels are accessed concurrently while in

a SAT the communication channels are accessed sequentially. The total cost for a CAT topology is the maximum of the cluster communication costs since the overall communication cost is limited by the slowest cluster. On the other hand, the total cost for a SAT topology is the sum of the cluster communication costs due to the sequential nature of the communication. Below we present some examples of SAT and CAT topologies:

$$T_{comm}\,[\textit{1-D}] = max_i\,\{T_{comm}\,[C_i, \textit{1-D}]\} \qquad \text{(Eq.3.7)}$$

$$T_{comm}\,[\textit{ring}] = sum_i\,\{T_{comm}\,[C_i, \textit{ring}]\}$$

$$T_{comm}\,[\textit{tree}] = T_{comm}\,[C_{root}, \textit{tree}] + max_{i \in \text{children}}\,\{T_{comm}\,[C_i, \textit{tree}]\}$$

$$T_{comm}\,[\textit{broadcast}] = sum_i\,\{T_{comm}\,[C_i, \textit{broadcast}]\,(b, P_T)*P_i\}/P_T$$

The *1-D* is an example of a CAT topology and the *ring* a SAT topology. The *tree* topology is more complicated. It has both concurrent communication (e.g., the children communicate simultaneously), and sequential communication (e.g., communication is ring-like from the leaves to the root). CAT topologies have a much greater potential for exploiting the additional communication bandwidth available in processor clusters and have better scaling properties. One notable exception is the *broadcast* topology.

The *broadcast* topology is a CAT but is complicated by the fact that all processors communicate with a single master processor. The absence of locality means that the communication cost cannot be characterized as a simple function of the individual communication costs within each cluster. We have observed empirically that for *broadcast* the total communication cost depends on the total number of processors P_T, and in a manner that depends on the number of processors contributed by each cluster. We compute the total communication cost as a weighted average based on the number of processors P_i contributed by each cluster C_i. This approximation turns out to be accurate in practice. This function has the property that the overall communication cost function converges to the communication cost function of the cluster that contributes the largest number of proces-

sors as the number of processors in this cluster is increased. This approximation makes the *broadcast* look more like a SAT topology in terms of performance properties.

The benefit of this communication model is that very accurate topology-specific communication costs can be estimated. We show that estimating these costs is key to effective scheduling. Once these cost functions are constructed they are stored in a configuration database where they are used in the scheduling process.

3.1.3 Resource Availability

Because the metasystem environment is shared, both communication bandwidth and processing resources may be committed to other users. We present a model for resource availability that accounts for resource sharing. A complete implementation of this model is outside the scope of this dissertation. We have implemented a useful subset of this model and discuss the implementation more fully in Chapter 5. Resource availability is implemented on top of existing operating system facilities and is limited by what the underlying operating system can provide.

The availability of computation cycles is based on a *reservation policy.* Processors may become unavailable due to reservation by other users. For example on a multicomputer, a user may allocate and reserve a portion of the machine. NX operating system facilities such as *pspart* and *cubeinfo* provide information about processor reservation for Intel multicomputers. In a workstation environment several systems have implemented reservation schemes that permit workstation owners to withdraw their machines from the shared set [35][52]. Machines also become unavailable if the amount of available computational resources is too little to be useful.

The availability of communication bandwidth is a more difficult problem. On multicomputers the amount of communication bandwidth is dependent on the size and location of the machine partition. On workstation clusters the available bandwidth depends on the current traffic profile. A network monitor can be used to estimate the available band-

width. Two possibilities for a network monitor are a *network tap* or the use of probe messages. The former is not likely to be applicable to a wider-area system where the use of taps compromises network security. Probe messages can be periodically sent out on the network and their travel time recorded to estimate bandwidth. This strategy could also be used to determine router costs dynamically. The reduced bandwidth estimate can be used to adjust the communication cost functions. Recall that these cost functions were benchmarked when the network was assumed to be lightly loaded and most of the peak network bandwidth was available.

However network traffic is notoriously bursty and unpredictable and it is not clear how useful this information would be in general. A better idea might be to provide a *guarantee policy* that serves as the dual of the reservation policy. A guarantee policy provides some guarantees on the available resources. For example suppose we are able to reserve all workstations in a processor cluster for some period of time and there are no other processors on the same network segment. We would then have the peak bandwidth available. Newer network technologies such as ATM [43] also offer the promise of dedicated bandwidth on a per connection basis. In the current implementation no available bandwidth information is collected. The *thermometer/thermostat* mechanism in the Legion system provides a way to specify the amount of computational resources that a single workstation can commit to a Legion user's application [34]. This is not enforced as a guarantee but such mechanisms may be useful in providing predictability in resource sharing.

Another factor that influences both the available computational resources and bandwidth is processor load. This is an issue for both workstation and multicomputer clusters since most multicomputer operating systems now support multiprogramming of individual processors. In the Unix environment processor load can be determined by a number of operating system facilities (*uptime, kmem*). We define load as the run-queue-length (RQL) over some time interval. This load index tends to be a good predictor of load in the

near future. In particular it can usually identify machines with long-running CPU-intensive jobs.

Processor load degrades both the available computational resources and effective communication bandwidth. Since a large part of the communication overhead is processor cost on workstation networks, the effective bandwidth is reduced by a loaded processor. The load measure can be used to degrade the power rating of a processor and the aggregate power of the cluster — for example a simple adjustment of 1/(RQL+1) can be made to the power rating. So if RQL=0, we expect the peak processor power, and if RQL=1, then we might expect to get 1/2 of the peak processor power since we are sharing the processor with another job. While such an adjustment appears to be better than no adjustment in some cases, we have determined that this adjustment is not dependable and can be fairly inaccurate. It is also clear that this load measure should be used to adjust the communication cost functions. Research into the quantitative impact of processor load on available computation and communication capacity is the subject of future work.

Another dimension to the resource sharing problem is memory. If a processor is running memory-intensive jobs, then the effective performance of the processor will be diminished due to paging. Normally there is a correlation between large memory demands and CPU cycle demands but not always. Consequently, memory availability is another variable that will impact resource availability. Treatment of memory availability is outside the scope of this dissertation.

We have implemented a simple scheme for dealing with resource sharing. All processors above a load threshold value are considered to be unavailable. This simple policy provides two benefits, it avoids highly loaded machines, and it allows computation and communication costs to be accurately determined. Accurate cost information is needed by partitioning and placement. If the load threshold is small enough then all available processors in a processor cluster can be treated as equal in computation power. But the threshold should be high enough to permit a sufficient number of processors to be marked available.

Resource availability is determined by the managers in Figure 3.1. The manager of a workstation-based cluster communicates periodically with each contained processor to collect load information. These managers also manage processor reservations if such a mechanism is provided. The manager of a multicomputer cluster can determine processor load information by using the operating system facilities described earlier. This information is then propagated as discussed in Section 3.1.1. An important issue outside the scope of this dissertation is fault tolerance for managers. If a processor upon which a manager is run goes down then another processor must be elected to become the manager. We have implemented a simpler scheme for resource availability described in Chapter 5.

An important issue is how the scheduling mechanism interacts with the managers. We have implemented a simple scheme suitable for a local-area environment described in Chapter 5. We now discuss alternatives that have better scaling properties and are more suitable for a wide-area environment. When a scheduling request for a data parallel computation arrives at the local cluster manager, a number of sites are probed to determine availability. The number of sites probed depends on an estimate of the amount of processing resources that the request will need — the estimate must be conservative. For example a large problem may require a large amount of resources so a sufficient number of sites must be contacted. Collecting all the resource information contained in a very large system is unnecessary for most applications. Using the resource availability of multiple sites would allow a single data parallel computation to be scheduled across multiple sites. Later we provide evidence in Chapter 6 that this may be feasible and also discuss some obstacles to achieving this in practice.

If we are willing to confine the scheduling decision to use machines within a single site then there is another alternative. Instead we send the scheduling request to a number of sites and have the sites run the scheduling algorithm in parallel. Again the number of sites would depend on an estimate of the amount of resources that are needed. Each site would return a bid based on how effective the site estimates it would be for the problem.

Effectiveness is measured as predicted completion time, a quantity that our scheduling method computes. The site with the smallest projected completion time would be selected.

3.2 Parallel Computation Model

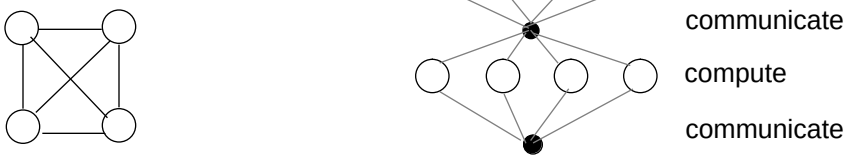
We have adopted a dynamic single-program-multiple-data (SPMD) model for data parallel computations. In the SPMD model a data parallel computation is performed by a set of identical tasks or *workers*, placed one per processor, each assigned a different portion of the data domain. Since workers are assigned one-to-one to processors we will often refer to processors, workers, or tasks interchangeably throughout this and subsequent chapters. The model is dynamic to allow tasks to be instantiated at runtime based on the processor selection. The SPMD model supports a computation granularity suitable for distributed-memory environments such as the metasystem. It has also been shown to be an effective implementation model for data parallel computations on multicomputers [41][57] and workstation networks [41].

Data parallel problems manipulate one or more data domains. We model the data domain as a collection of primitive data units or *PDU*s, where the *PDU* is the smallest unit of data decomposition. The *PDU* is problem and application specific. For example, the *PDU* might be a row, column, or block of a matrix in a matrix-based problem, a DNA sequence in a gene sequence matching problem [30], or a collection of particles in a particle simulation. The *PDU* is similar to the virtual processor [62] but may also arise from unstructured data domains. *PDU*s are assigned to workers during partitioning. Scheduling does not depend on the nature of the *PDU* but rather manipulates *PDU*s in the abstract.

Two views of the data parallel computation are provided to the scheduling framework — *task* view and *phase* view. In the task view, the computation is represented as a collection of communicating *workers* or processes in a static task graph, see Figure 3.7(a). SPMD computations are naturally expressed by the STG. An advantage of the STG is that it exposes important topology information that is needed by placement. On the other hand,

the task view encapsulates important information about the communication and computation structure of the problem. The phase view provides this information.

In the phase view, the computation is represented as a sequence of alternating computation and communication *phases* [56], see Figure 3.7(b). The dotted lines indicate that the workers are communicating together in some pattern, not necessarily a fan-in as depicted in Figure 3.7(b). Each worker participates in the execution of these phases. These phases are more tightly-coupled than the phases discussed in [66] which require data redistribution. A communication phase contains a synchronous communication executed by all processors. A computation phase contains only computation. Communication and computation phases may be overlapped. Most data parallel computations are iterative with the computation and communication phases repeating after some number of phases. This is known as a *cycle*.



a) Task view b) Phase view

Figure 3.7: Two views of a data parallel computation

The phase view provides important information that is needed by partitioning and placement. This information is provided by *callback* functions. The callbacks are a set of runtime functions that provide critical information about the communication and computation structure of the implementation.

3.2.1 Function Callbacks

The callbacks provide the minimal amount of information that is needed to support the partitioning and placement process. It is important to mention that the callbacks pro-

vide information about a particular implementation of a data parallel problem. A different implementation of the same problem may require different callback functions. In some cases conservative cost information can be used if callbacks are omitted. We present an implementation of the callbacks complete with function signatures in Chapter 5. For now we describe the callbacks in the abstract. Two callback functions refer to the computation as a whole:

- *numPDUs*
- *overlap*

The number of *PDU*s in the problem, *numPDUs*, is akin to the problem size. It may depend on any number of problem paramters. This callback is the same for all computation phases within a particular data parallel computation. The *overlap* callback is used to specify whether any computation and communication phases overlap in time. The current implementation supports the overlap of a single computation and communication phase.

Each computation phase has the following callbacks defined:

- *comp_complexity*
- *arch_cost*

The amount of computation performed on a *PDU* in a single cycle is known as the computation complexity, c*omp_complexity.* It has two components: the number of instructions executed on a per *PDU* basis, and the number of instructions executed that do not depend on the *PDU*. The first component is typically a function of problem parameters and the second is often small enough to omit. The former provides the average number of instructions executed on a *PDU* in a single cycle. It can be determined by summing up the total number of instructions executed over all *PDU*s over all cycles and then dividing by the number of *PDU*s and the number of cycles. In most cases this reduces to a simple function as we will show. The *comp_complexity* is architecture-independent. Multiplying the *comp_complexity* times the peak instruction rate (μsec/instruction) for a given architecture

provides a best-case estimate of the expected execution time for a *PDU*. This formulation ignores memory and caching effects, paging and other architecture-dependent costs. Nevertheless, we have found it to be a good estimator. A better estimator is based on the *arch_cost* callback.

The architecture-specific execution costs associated with *comp_complexity* are captured by *arch_cost,* provided in units of μsec/instruction. It also has two components corresponding to the architecture-specific *PDU* dependent and independent costs respectively. The *arch_cost* contains an entry for each processor type in the target metasystem. To obtain the *arch_cost*, the *sequential* application code (i.e., the parallel code running on one processor) must be benchmarked on each processor type and the total *PDU* execution time divided by the total number of instructions executed. A much more accurate estimate of the expected execution time for a *PDU* becomes *arch_cost* times *comp_complexity.* It is more accurate because *arch_cost* includes memory and caching costs. We have observed that the *arch_cost* may be sensitive to problem-size due to memory and cache effects and a range of *arch_cost* values can be specified. We give an example of this in Chapter 7. An alternative is to form the *arch_cost* as an average over a range of problem sizes.

Each communication phase has the following callbacks defined:

- *topology*
- *comm_complexity*

The topology refers to the communication topology of the communication phase. The amount of communication between tasks is known as the communication complexity, *comm_complexity.* It is the average number of bytes transmitted by a worker in a single communication during a single cycle of the communication phase. It can be determined by summing up the total bytes transmitted over all cycles and then dividing by the number of cycles. In most cases the *comm_complexity* also reduces to a simple function. Similar to *comp_complexity,* it has two components: the number of bytes transmitted per *PDU* and

the number of bytes transmitted that are independent of the number of *PDU*s. It is used to determine the parameter *b* in the communication cost equations.

In some cases the callbacks may depend on other parameters unknown until runtime such as the number of processors used. These parameters are passed automatically to each callback function and may be used in the callback implementation. We describe the implementation of callback functions later in Chapter 5.

Among the computation and communication phases, two phases are distinguished. The *dominant* computation phase has the largest computation complexity, while the *dominant* communication phase has the largest communication complexity. The dominant phases may depend on problem parameters and we have extended the callback mechanism to provide this information. We have implemented two strategies for using the callbacks in guiding the partitioning and placement process. The simplest and cheapest uses the callbacks associated with the dominant phases only. The other is more accurate and expensive and uses the callbacks associated with all phases.

An example that illustrates the callbacks for a regular *N*x*N* five-point stencil computation for a PDE solver: $-u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} + 4u_{i,j} = 0, i, j = 1, \ldots, N$ is given in Figure 3.8 (the *arch_cost* is omitted). The PDE solver uses Jacobi's method. These are functions that return the values indicated. For *comp_complexity* we show only the *PDU* dependent cost and for *comm_complexity* we show only the *PDU* independent message size. This computation has been implemented using a block-row decomposition of the grid as depicted in Figure 3.8(a). In this implementation the *PDU* is a single row and the processors are arranged in a *1-D* communication topology. The stencil computation is iterative and consists of two dominant phases: a *1-D* communication to exchange north and south borders, and a simple computation phase that computes the function value at each grid point to be the average of its neighbors.

Notice that the callback functions may depend on problem parameters (e.g., *N*) that are unknown until runtime. The callbacks for the computation and communication com-

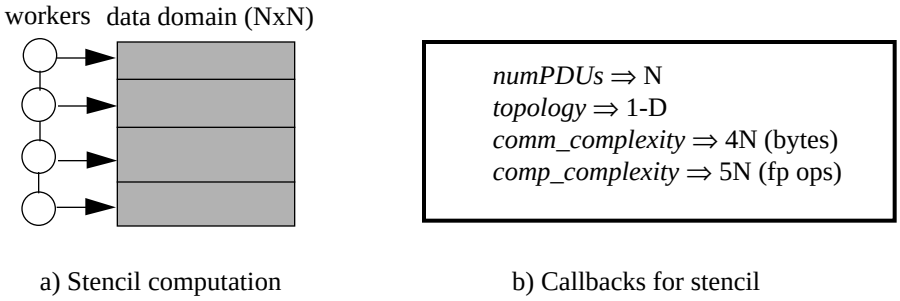


Figure 3.8: Example: 1-D stencil computation

plexity allow an estimate of the computation granularity to be computed at runtime. This estimate is used to determine the number of processors to use. The topology is used to select the appropriate communication function. The computation complexity is also used to determine a decomposition of the data domain, i.e., the number of *PDU*s to be assigned to each worker.

The callback mechanism is very powerful and can be applied to data parallel computations less regular than the five-point stencil. Since the callbacks may be arbitrary and complex functions and may depend on any number of problem parameters, they can handle some data-dependent computations by pre-processing the data domain. For example, the computation complexity for a sparse matrix problem typically depends on the non-zero structure of the matrix. But a simple callback can be written to capture this dependence. We have done this for a finite-element problem presented in Chapter 7. Similarly for irregular computations that are run repeatedly such as a global climate model code [60], the callbacks may be based on the statistics generated from previous runs.

For irregular or control-dependent data parallel computations, off-line benchmarking of the sequential code may be needed to determine average values for *comp_complexity* and *comm_complexity.* The instruction counts and message sizes needed for these callbacks can be determined by inserting probes into the code. We have done this for the finite-element and biological sequence codes presented in Chapter 7. We have already discussed that the *arch_cost* callback requires architecture-specific benchmarking.

Fortunately, *comp_complexity* and *comm_complexity* are architecture-independent and need not be benchmarked on each architecture type.

We present an implementation of callbacks later in Chapter 5 and present the callbacks for a number of data parallel computations in Chapter 7.

3.2.2 Data Decomposition

In a heterogeneous environment workers may be assigned different numbers of *PDU*s in order to balance the computational load. The decomposition information is contained in a structure known as the *partition_map* that is defined as follows:

A_i = number of *PDU*s assigned to the worker on processor p_i
$\Sigma A_i = numPDUs$

The *partition_map* has an entry for each processor or worker and the association of its entries to workers may be *topology-dependent*, see Figure 3.9. The topology-dependence reflects the data locality relationships in the problem. Data locality means that elements of the data domain have some relationship to each other. For example in the *1-D* stencil problem of Figure 3.8, points on the grid are coupled to their neighbors. This information is needed when the data domain is decomposed to the workers. For example, a 100x100 grid might be decomposed across four workers as shown in Figure 3.9(a), worker 1 gets the first 20 *PDU*s or rows, worker 2 gets the next 30 *PDU*s, and so on. If we assume the workers are arranged in a *1-D* topology with worker 1 at the top, followed by worker 2, ... and so on, then the *1-D* communication preserves the data locality relationships. On the other hand in Figure 3.9(d) there are no data locality relationships and the data decomposition is not constrained. We will see both types of decompositions later in Chapter 7.

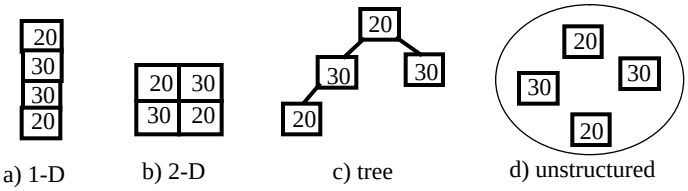


Figure 3.9: Topology-dependent partition_map (numPDUs = 100)

The *partition_map* is a logical decomposition of the data domain and is computed at runtime by partitioning. The implementation is responsible for using the *partition_map* in a manner appropriate to the problem. For example, an out-of-core implementation for very large grids might simply pass the *partition_map* to the workers and have them acquire their portion of the grid individually from disk. In Chapter 7, we sketch an in-core implementation of the stencil problem in which the main program uses the *partition_map* to physically decompose the grid and then distributes pieces of the grid to the appropriate workers.

Decomposing the data domain from the *partition_map* must satisfy load balance and data locality requirements. If the amount of computation per *PDU* is the same for all *PDU*s then achieving *static* load balance is straightforward. The number of *PDU*s assigned must only match the entries of the *partition_map*. The problem becomes slightly more complicated if there are locality relationships since this imposes restrictions on the assignment. But both of these problems are easily solved for most regular problems.

If the amount of computation per *PDU* is not the same for all *PDU*s then achieving load balance can be more difficult. If there are no locality relationships then several strategies can be used. Randomizing the data domain tends to work well for large problems. Exploiting problem knowledge can also be effective. For example, in Gaussian elimination we decompose the matrix by a cyclic interleaving of rows to provide load balance. If there are data locality relationships then the data decomposition problem can be difficult and problem knowledge must be used. In Chapter 7, we present data parallel computations that fall into each category.

A decomposition that satisfies load balance can be easily expressed. Define $comp_i$ to be sum of the execution times for the *PDU*s assigned to worker *i* and $\overline{comp}$ as the average execution time over all *PDU*s in the problem. The *partition_map* entry can be interpreted as the percentage of work to be assigned to worker *i*.

Then the following must hold for all workers:

$$comp_i \approx \frac{A_i}{numPDUs} [\overline{comp} \cdot numPDUs] \Rightarrow$$

$$comp_i \approx A_i \overline{comp}$$

The first term $\frac{A_i}{numPDUs}$ is the work percentage that is to be assigned to worker *i* and the second term in braces is the total amount of work in the problem. Note that when the *PDU* cost is the same for all *PDU*s this relation holds trivially. The physical decomposition must satisfy the relation above in order to achieve load balance.

If the amount of computation per *PDU* varies at runtime in an unpredictable fashion then a load imbalance may arise and some form of *dynamic* repartitioning is needed. This topic is addressed in Chapter 8.

3.2.3 Multiple Data Parallel Computations

A problem may contain several data parallel computations. Different data parallel computations may operate on different data domains, may require data redistribution, and may be coupled to each other. For example, the finite-element problem that we present later contains two coupled data parallel computations that operate on two different data domains though no data redistribution is needed.

Each data parallel computation may be scheduled individually. The current implementation can handle multiple *sequential* data parallel computations. Gaussian elimination and the finite-element problem are two examples. The scheduling of *concurrent* data parallel computations is a more difficult problem. One possibility is to extend the notion of dominant phases to dominant computations. Dominant computations would be scheduled first and allocated the best available resources. The scheduling of these problems is outside the scope of this dissertation.

A single data parallel computation will be scheduled at a time and it is the responsibility of the implementation to indicate the order. The implementation must also perform

any data redistributions that are needed between execution of these data parallel computations. A single *partition_map* is computed for each data parallel computation that is scheduled.

3.2.4 SPMD-like Data Parallel Computations

We have extended the SPMD model to include a common model for implementing data parallel computations in which the SPMD tasks may not identical. Consider a fan-in/ fan-out *tree* where the leaves are performing the computation (i.e., the workers), and the interior nodes are responsible for communicating results up and down the tree only, see Figure 3.10. This allows more effective overlap of computation and communication. The leaf computations are overlapped with interior node communications. The leaves and the interior nodes execute different SPMD programs. We refer to this organization as a *hybrid-tree* and it is specified via the topology callback. The framework implementation is more complex for *hybrid-tree* — the *partition_map* applies only to the leaves, and the placement of tasks becomes more difficult since interior and leaf nodes must be treated differently. An example of this type of problem is the biological sequence comparison, complib, discussed in Chapter 7.

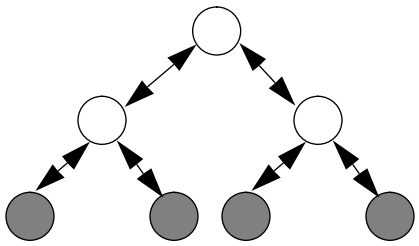


Figure 3.10: Hybrid-tree topology

3.2.5 Compiler Support

The SPMD computation model does not assume a particular language model. It is assumed that an SPMD worker implementation together with the callback functions are

provided. The details of the programmer interface and a callback implementation are discussed in Chapter 5.

Advanced compilation techniques can be used with appropriate language constructs to generate some of the callbacks for many regular problems. For example, it is easy to see how the callbacks for stencil might be generated. Such language support has been proposed in a integrated data parallel control parallel language called Braid [94]. Braid supports the explicit specification of application communication topology, dominant computations, and a concept known as subset data parallelism which provides information that is similar to the *PDU*.

However for irregular, control- or data-dependent computations it is likely that the domain programmer will have to write some callback functions by-hand. If this is the case, it may be possible to simplify this task by providing libraries of callbacks for well-known problem types. The programmer could extend these template callbacks in a manner appropriate to the problem at hand. For example, a set of generic callbacks for stencil-based problems could be provided. For a stencil-based application such as an image processing problem or iterative PDE solver, the stencil callbacks could be tailored to fit the problem. The development of callback libraries is the subject of future work.

3.2.6 Limitations

The model does not capture a number of problem classes. A class of problems in which *PDU*s are shifted between processors during the course of execution may require dynamic repartitioning of the data domain to preserve load balance. Examples of these problems include molecular dynamics and particle-in-cell codes. Our model is not incompatible with dynamic partitioning but it is outside the scope of this dissertation. Another problem class is one in which the workload is generated in a stochastic fashion. Benchmarking the application will not necessarily be helpful in determining the callbacks since

the problem characteristics may depend on random events. An example of this type of application would be certain parallel discrete event simulations.

In this chapter we have presented a model for representing metasystem resources and a model for representing parallel computations. These models define the information needed to construct cost functions for computation and communication. These models form the cornerstone of the scheduling framework described in the next chapter.

Chapter 4 Partitioning and Placement

This chapter introduces the partitioning and placement problem and several promising heuristics. The objective of partitioning and placement is to achieve reduced completion time for the data parallel computation. Partitioning estimates the best subset of available processors to use based on computation granularity and a heterogeneous decomposition of the data domain based on load balance. We formulate partitioning as a mathematical optimization problem and present two effective heuristics. Placement assigns workers to the selected subset of processors in a manner that reduces the communication overhead. Partitioning and placement are solved together in the scheduling framework. Both partitioning and placement rely on a set of runtime cost functions for computation and communication that have been constructed from system resource and program information.

4.1 The Partitioning Problem

Partitioning divides the problem across a set of processors at an appropriate grain size. If too many processors are selected, the computation granularity will be too small and communication overhead may dominate the benefit of increased parallelism. On the other hand if too few processors are selected, the computation granularity will be too large and insufficient parallelism has been exploited. Selecting the processors to use from among the available set is known as *processor selection*. A worker is assigned to each

selected processor. The optimum processor selection depends on characteristics of the problem and of the available processing resources.

For a selected set of processors, partitioning also determines a load balanced decomposition of the data domain. Recall that the decomposition information is kept in a structure known as the *partition_map*. In a load balanced decomposition of the data domain, all processors or workers will finish at the same time. A load balanced decomposition with an appropriate computation granularity leads to reduced completion time.

Partitioning and placement are performed at runtime given the available processing resources. In the current implementation, partitioning and placement are done *statically* at runtime. We believe dynamic repartitioning in the event of load imbalance could be accommodated within the framework and this is addressed later in Chapter 8. We will use the following notation throughout this chapter:

p_i = a particular processor
A_i = number of PDUs assigned to processor p_i
V_j = number of available processors within cluster C_j
P_j = number of processors selected for C_j
w_i = relative processor weight for i^{th} processor (problem-specific)
m = number of clusters
g() = the amount of computation as a function of A
x_i = PDU independent cost constant for i^{th} processor
y_i = PDU dependent cost constant for i^{th} processor
T_c = per cycle elapsed time
DP = set of all data parallel computations for the problem
d = a particular data parallel computation
$T_{startup}$ = start-up overhead
T_{comm} = per cycle communication cost
T_{comp} = per cycle computation cost

We begin with a discussion of data domain decomposition and show how a load balanced decomposition is computed for a collection of heterogeneous processors. We also show that a load balanced decomposition for a fixed set of processors is optimal. Fol-

lowing this we discuss the processor selection process. Processor selection assumes a load balanced decomposition for each set of candidate processors.

4.1.1 Data Domain Decomposition

We compute a load balanced decomposition for each candidate *processor configuration* explored by the scheduling method. A processor configuration is a set of processors P_j ($0 \leq P_j \leq V_j$, j=1 to m), where V_j is the number of processors available within C_j. The data domain decomposition is based on the amount of time spent in computation. Recall that in Chapter 3, the communication costs experienced by all processors or workers is the same for synchronous communications. So communication need not be considered for load balance. We present a method for decomposing the data domain based on the dominant computation phase.

The amount of time spent in a single cycle of the dominant computation phase, denoted by T_{comp}, is defined as follows (shown for a processor p_i):

$$T_{comp}\,[p_i] = comp\_complexity * arch\_cost\,(p_i) * g(A_i) \qquad \text{(Eq.4.1)}$$

The computation time depends on the problem and processor characteristics and on number of *PDU*s, A_i, given to p_i. In general the dependence on A_i may be an arbitrary function g of A_i. At runtime when the problem parameters are known, the callbacks in (Eq.4.1) are invoked for *comp_complexity* (number of instructions per *PDU*) and *arch_cost* (time per instruction) and the form for T_{comp} becomes:

$$T_{comp}\,[p_i] = x_i + y_i g(A_i) \qquad \text{(Eq.4.2)}$$

where x_i and y_i are constants formed by multiplying the respective *PDU* dependent and *PDU* independent terms for the callbacks in (Eq.4.1). Recall that both *comp_complexity* and *arch_cost* have a *PDU* dependent and *PDU* independent component and that *arch_cost* will reflect architecture-specific costs such as memory access overhead. For example, consider the callbacks for the stencil computation in Figure 3.8 for N=100. Suppose the *arch_cost* on p_i is 0.1 μsec for both the *PDU* independent and *PDU* dependent

execution time and the *comp_complexity* is *5N* for the *PDU* dependent part of the computation and 25 instructions for the *PDU* independent part of the computation. The value for x_i becomes (25)*0.1 or 2.5 μsec and the value for y_i becomes (5*100)*0.1 or 50 μsec. The terms in parenthesis are the total number of instructions.

Load balance requires that T_{comp} be the same for all processors (*P* total processors):

$$x_1 + y_1 g(A_1) = x_2 + y_2 g(A_2) = \ldots x_P + y_P g(A_P) \qquad \text{(Eq.4.3)}$$

$$\text{subject to } \sum A_i = numPDUs$$

If *g* is non-linear then this is a difficult system to solve and iterative methods must be used. In practice however *g* is linear for SPMD computations in which the same computation is performed on each data element (i.e., *PDU*) independently. If *g* is linear, we can combine this equation with the equality constraint to easily compute the *partition_map*. To do this we first define w_i which is the relative processor weight for p_i based on *arch_cost* (*k* ranges over all selected processors):

$$w_i = \frac{max\{y_k\}}{y_i}$$

A smaller y_i means a larger weight since y_i is in units of time per instruction. The equation for the *partition_map* is easily expressed as a function of the relative processor weights:

$$A_i = \left(\sum_k \frac{w_i}{w_k} \right) \cdot \left[numPDUs - \sum_k \frac{x_k - x_i}{y_k} \right] \qquad \text{(Eq.4.4)}$$

A special case of (Eq.4.4) occurs when the *PDU* independent cost is 0 (i.e., $x_i = 0$):

$$A_i = \sum_k \frac{w_i}{w_k} \cdot numPDUs \qquad \text{(Eq.4.5)}$$

This equation has the property that *faster* processors will receive a greater share of the data domain and processors in the same cluster will receive an equal share since the associated w_i will be the same[1]. Faster processors do not necessarily imply processors with the highest peak rates, but processors that can perform this computation most efficiently. Since A_i must be integral, the individual entries in the *partition_map* must be rounded to the nearest integer. This will leave some *PDU*s unaccounted for so we assign the left-over *PDU*s to the fastest processors. We do not account for left-over *PDU*s in the above equations.

An alternate strategy is to use the callbacks associated with all computation phases. The amount of time spent in all computation phases is the following:

$$T_{comp}[p_i] = \sum_{phases} x_i + y_i g(A_i) \qquad \text{(Eq.4.6)}$$

If all computation phases are linear in A_i then we can rewrite (Eq.4.4) as follows:

$$A_i = \left(\sum_k \frac{w_i}{w_k} \right) \cdot \left[numPDUs - \sum_k \frac{X_k - X_i}{Y_k} \right], \; w_i = \frac{max\{Y_k\}}{Y_i} \qquad \text{(Eq.4.7)}$$

where X_i is the sum of all x_i and Y_i is the sum of all y_i associated with each computation phase.

It is well-known that load balance is a necessary condition for achieving minimum completion time for synchronous SPMD computations. The *partition_map* computed by (Eq.4.5) gives load balance for a non-integral *partition_map*. However, the integer solution we obtained by rounding and assigning the extra *PDU*s to the fastest processors is a good heuristic for reducing completion time. Since a processor may receive at most one additional *PDU* in the integer solution, the percent increase in execution time with respect

1. This will not be the case when processor load is considered and w_i may be reduced.

to the optimal load balance decomposition is at most 1/*NumPDUs* under assumptions of linearity.

If the message size depends on A_i then it is possible that the optimal *partition_map* does not necessarily load balance the processors. This situation might arise if a cluster has very different computation and communication capacities. For example if a cluster has very fast processors with poor communication bandwidth then it may be better to off-load *PDU*s to a cluster that may have slower processors but with a greater communication bandwidth. In this event computing the *partition_map* that load balances the processors may be sub-optimal. However, the experimental results indicate that for two problems in this class, computational load balance results in reduced elapsed time.

Load balance guarantees that T_{comp} will be the same for all processors or workers and we drop the p_i subscript on T_{comp} in the remainder of this chapter. Computing the *partition_map* using either the dominant computation phase or all computation phases is performed for a particular processor configuration. Choosing the number of processors to use, P_j for each C_j (i.e., to determine the range for k) is the subject of processor selection, discussed next.

4.1.2 Processor Selection

Nearly all parallel computations reach a point of diminishing returns with respect to the number of processors that can be used effectively. At that point we have achieved the best computation grain for the problem. Locating this point is difficult when the processors are homogeneous and is even more difficult when the processors are heterogeneous. We analyze this problem and present several heuristics. The heuristics are guided by runtime cost estimation that use information provided by the callback functions.

We define the elapsed time $T_{elapsed}$ for a problem that contains a number of sequential data parallel computations *DP* as follows:

$$T_{elapsed} = T_{startup} + \sum_{d \in DP} \sum_{i=1}^{cycles[d]} T_c[d, i] \qquad \text{(Eq.4.8)}$$

The start-up overhead $T_{startup}$ may include any initial data distribution or problem setup costs. The amount of time spent in the i^{th} iteration or cycle of the d^{th} data parallel computation is denoted by $T_c[d, i]$ and the number of cycles is denoted by $cycles[d]$. We denote $T_c[d]$ as the average value of $T_c[d, i]$ over all cycles in d and rewrite (Eq.4.8) as:

$$T_{elapsed} = T_{startup} + \sum_{d \in DP} T_c[d] \cdot cycles[d] \qquad \text{(Eq.4.9)}$$

If $T_{startup}$ is small relative to the elapsed time, then minimizing $T_{elapsed}$ can be achieved by minimizing the sum in (Eq.4.9). Minimizing this sum can be achieved by minimizing $T_c[d]$ for each data parallel computation. We now assume that the problem contains only one data parallel computation and the d subscript may be dropped. This assumption is made in order to simplify the remainder of this chapter. All of the results we present apply to the more general case as well unless data needs to be redistributed between successive data parallel computations. In this case, a cost function that characterizes the cost of data redistribution is needed. This is outside the scope of the dissertation.

Minimizing $T_{elapsed}$ is achieved by minimizing T_c, the average per cycle execution cost. T_c is a function of the per cycle computation and communication costs for each computation and communication phase (the superscript indicates the phase):

$$T_c = f(T_{comp}^{1}, T_{comp}^{2}, \ldots T_{comm}^{1}, T_{comm}^{2}, \ldots)$$

In general this may be a complex function due to the possibility that multiple computation and communication phases overlap in time. We make the assumption that only the dominant computation and communication phases are overlapped to limit the different formulations of T_c that need to be handled by the framework implementation. Additional formulations can be easily added to the implementation. We denote T_{comp} as the total computation cost and T_{comm} as the total communication cost components of T_c. We consider

two methods for estimating T_{comp} and T_{comm}: (1) computation and communication costs are determined using dominant phases only and (2) computation and communication costs are determined by summing all phases. In the current implementation for (2) there is no overlap of computation and communication permitted by the implementation. This could be supported with a more complex *overlap* callback specification.

We consider two common forms for T_c depending on whether computation and communication are overlapped:

$$T_c = T_{comp} + T_{comm} \text{ or} \qquad \text{(Eq.4.10)}$$
$$T_c = max\ \{T_{comp}, T_{comm}\} \text{ if overlap}$$

We show later in this section how T_c can be easily constructed at runtime using program and resource information.

The minimization of T_c requires the solution of an inequality-constrained, non-linear, integer programming problem. This function may also be non-convex. The potential presence of *max* as shown in (Eq.4.10) means that iterative, gradient-based methods cannot be used since the objective function does not have continuous derivatives. There may also be discontinuities due to *arch_cost* changing for different problem sizes. Consider the first form for T_c in (Eq.4.10) and assume that the computation and communication costs of the dominant phases are used. The form for this T_c is given below:

$$T_c = T_{comp} + T_{comm}$$

$$T_{comp} = x_i + y_i A_i \text{ [via (Eq.4.2) for any } i\text{]}$$

$$= x_i + y_i \sum_j \frac{w_i}{w_k} \cdot numPDUs \qquad \text{[via (Eq.4.5) substituting for } A_i\text{]}$$

Observe that this is a non-linear function in the number of processors (the w_k correspond to k selected processors). The communication cost T_{comm} is defined by (Eq.3.7). The form for T_c becomes:

$$T_c = x_i + y_i \sum_k \frac{w_i}{w_k} \cdot numPDUs \quad + F\{T_{comm}\ [C_j, 1\text{-}D], \text{ for all selected } C_j\}$$

where x_i, y_i, w_i are all constants. There are additional constants depending on the precise

form of the communication cost function. T_c is the same for any value of i since T_{comp} is the same for all processors (under load balance) and T_{comm} is the same under our assumptions of synchronous communication. The mathematical optimization problem is to minimize T_c subject to:

$0 \leq P_j \leq V_j$, P_j integral.

The additional constraints on A_i given in (Eq.4.3) are satisfied by the substitution of (Eq.4.5) above.

T_c is non-linear in the number of processors. This non-linearity may arise from several sources — T_{comp} via (Eq.4.5) or from the communication functions f (Eq.3.3) or F (Eq.3.6). T_c may also be non-convex due to max from (Eq.4.10) or from a max that appears due to a CAT communication topology (Eq.3.7). Thus, the minimization of T_c is a hard problem to solve optimally.

We have developed two heuristics that have worked well in simulation studies and when applied to several real data parallel computations. These heuristics attempt to locate a minimum for T_c by searching a portion of the solution space. The entire solution space is exponential in the number of clusters and processors.

We present several graphs for different formulations of T_c to help motivate the heuristics. First consider the simplest case — a single processor cluster with a communication cost function f that is linear in the number of processors, a message size b that does not depend on the number of processors, and no computation or communication overlap. This particular T_c corresponds to the *1-D* stencil problem on a workstation cluster. We get an equation for T_c that is the result of combining all of the constants for T_{comp} and T_{comm} from the equation for T_c given above. We omit the definitions for these constants which we denote by a_1, a_2, ... as the analysis does not depend on them.

If the message size depends on the number of processors, the same form for T_c results. This graph is plotted in Figure 4.1(a) and observe the predictable parabolic shape for T_c. Note that when P=1, no communication cost is paid. The minimum point is obtained

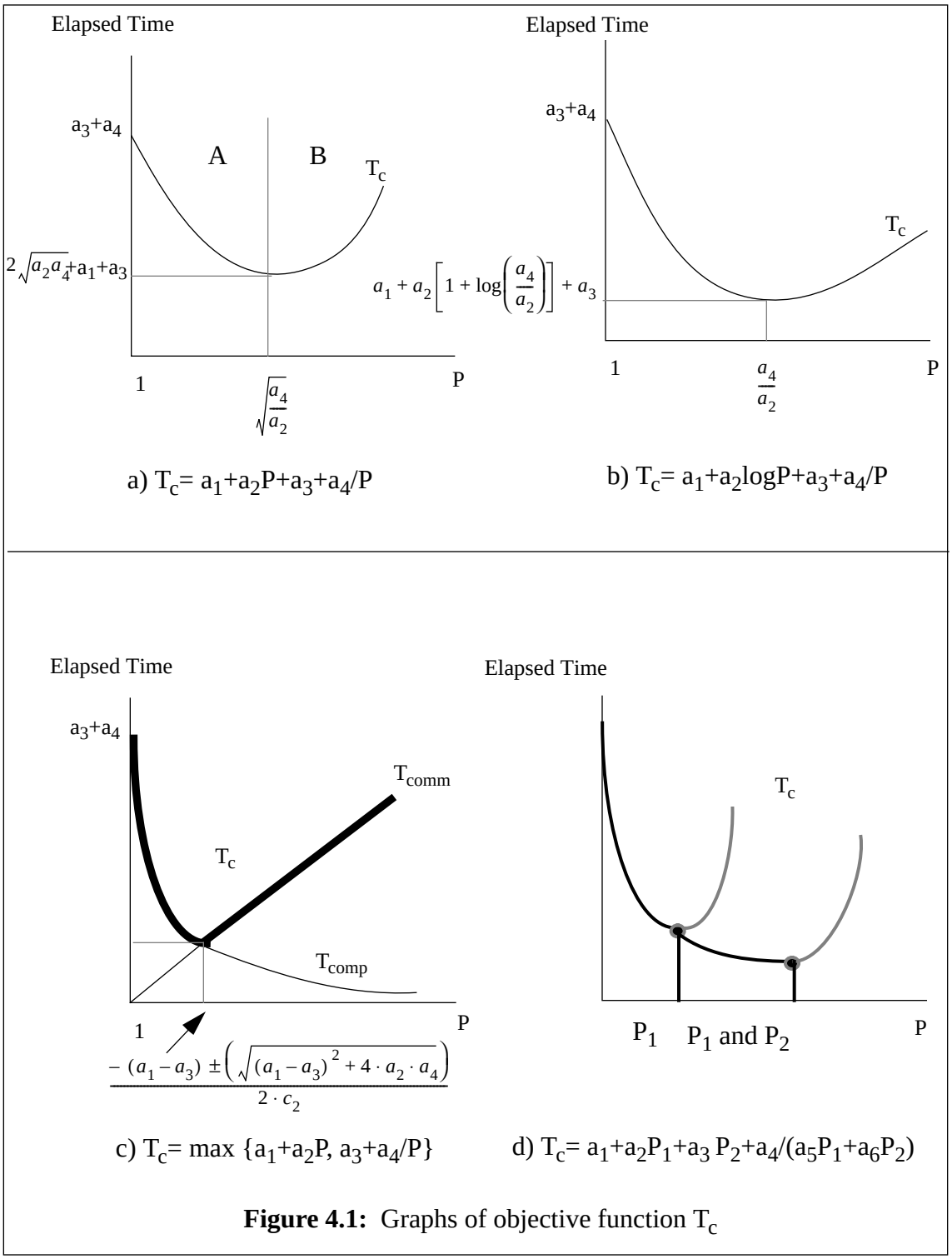


Figure 4.1: Graphs of objective function T_c

by differentiating T_c and setting the right-hand-side to 0. In region A, the computation granularity is too large and in region B the computation granularity is too small. We have shown the common case where T_c is unimodal. It is possible that T_c will have local minima if pro-

cessor loads differ within the cluster, or there is a *max* in the formulation for T_c, or if the *PDU* execution cost is very sensitive to problem size due to memory and caching costs.

Next suppose that the communication cost function *f* is *logarithmic* in the number of processors as is common for tree communications. In Figure 4.1(b) the same parabolic shape for T_c is observed but the minimum occurs at a different point. If the message size depends on the number of processors then a slightly more complex form for T_c results.

A more interesting case occurs when computation and communication are overlapped. Suppose that the communication cost function *f* is linear and computation and communication are fully overlapped. In Figure 4.1(c) the presence of *max* introduces a discontinuity in the graph for T_c. We have plotted T_{comp}, T_{comm}, and T_c on the same axis, with T_c being the portion of T_{comp} and T_{comm} in bold. The minimum occurs at the point where T_{comp} and T_{comm} are equal.

Now suppose that the number of processor clusters is > 1. Consider the simplest case of two processor clusters C_1 and C_2, linear communication costs in both clusters, and the dominant communication topology is a synchronous access topology (SAT) such that communication costs are additive. In this case, T_c has two dependent variables, P_1 and P_2, the number of processors selected in each cluster. Suppose that the processors in C_1 are a better choice for this computation and would yield a smaller elapsed time than if processors in C_2 were used instead. In this instance we would use all processors in C_1 before using any processors in C_2. This can be generalized to any number of clusters. We plot T_c as shown in Figure 4.1(d). Along the x-axis, we begin with processors in C_1 for $P_1 = 1 .. V_1$, where V_1 is the total number of processors available in C_1. This portion of the graph is the same as in Figure 4.1(a). Depending on the problem and the number of available processors in C_1, the minimum elapsed time may fall within this portion of the graph. The dotted line indicates that this may be the case. However, if the computation granularity is large then processors in C_2 may also be used and this is indicated by the next portion of the graph. The junction at which the next portion of the graph begins also depends on the problem and

cluster characteristics. In the region labelled P_1 and P_2, all processors in P_1 are used together with $P_2 = 1 .. V_2$. Additional processor clusters would be handled in the same fashion. It is also possible that the minimum may occur at a point in which processors in both C_1 and C_2 are used, but P_1 is less than V_1.

In general we cannot rely on standard minimization procedures since T_c may have discontinuities. Furthermore, the majority of these methods are iterative which may require substantial runtime overhead to reach a converged solution. Instead, we have developed two heuristics that are not guaranteed to find the optimal solution, but have proven to be effective and have a small and predictable runtime cost. The heuristics are based on the technique discussed for Figure 4.1(d) above, *cluster ordering*.

It is not possible to explore all processor configurations since the space is exponential in both the number of processors and clusters. Cluster ordering is used to reduce the search space by considering processors belonging to the best clusters first. The best clusters depend on the problem. A cluster with a large communication capacity might be a better choice for a tightly-coupled problem with a large amount of communication. On the other hand, a cluster with a large computation capacity might be better for a problem with a large computation granularity. Some problems will also perform better on certain machines based on architectural characteristics and may even perform better on different machines for different problem sizes. Cluster ordering exploits machine-problem affinities by considering both computation and communication performance.

We describe two heuristics for processor selection, H_1 and H_2, that have yielded promising results. H_2 is a special case of H_1. Both heuristics explore a series of processor configurations in an attempt to achieve a minimum T_c, hence minimized completion time. For each configuration explored, T_c is computed via (Eq.4.10). To do this we first compute the *partition_map* via (Eq.4.5). Once the data decomposition is determined, we can compute T_{comp} (Eq.4.1) and T_{comm} (Eq.3.7) easily by invoking the callbacks and selecting the appropriate communication function. All of these computations are simple and can be per-

formed efficiently at runtime. For a given configuration, the placement heuristics are used to determine task placement and the expected communication costs that result using this placement are included in T_{comm}. Placement is discussed in the next section. The general form of the processor selection heuristics is shown in Figure 4.2.

1. Order processor clusters
2. Repeat
    3. Select next candidate processor configuration
    4. Compute partition_map
    5. Compute T_{comp}, T_{comm} and T_c
    6. If T_c is best, store this processor configuration
7. Until done

Figure 4.2: Processor selection algorithm

Heuristic H_1

Heuristic H_1 has been designed for environments in which computation and communication capacities may vary throughout the metasystem. Because communication capacities may be different, a simple cluster ordering strategy based solely on computation power will not always work well. For example, consider that a slow network of very fast machines such as a DEC-Alpha cluster might be chosen over a Paragon partition because the DEC-Alpha is faster than the i860. Clearly this may be a poor choice for some tightly-coupled parallel computations.

A metric for cluster ordering must consider both computation and communication cost. A real measure of computation and communication cost is provided by T_c. For each cluster we compute the smallest T_c value obtained using only processors in this cluster. The clusters with the smallest T_c value are chosen first. The ordering algorithm performs a binary search on the processors in C_i on the interval [1 .. V_i] to find the smallest T_c. If there are m clusters and P_{max} is the largest number of processors in a cluster then the worst-case complexity of cluster ordering is θ ($mlogP_{max}$). If there is a single minima for T_c within each cluster then this procedure is guaranteed to find it. If there are multiple minima then

this method becomes a heuristic that is not guaranteed to find the minimum, but it has worked well in simulation and experimental studies.

Cluster ordering does not consider routing and conversion costs between clusters. In local-area environments where routing costs are similar between clusters this is reasonable. In a wide-area environment where routing costs may differ by orders of magnitude, routing costs will have to be included if clusters in multiple sites are to be considered for the same problem. For this reason we would expect the performance of H_1 to fall off in the wide-area setting. Cluster ordering in a wide-area environment is the subject of future work.

A two-phase strategy is adopted for exploring the processor configurations, see Figure 4.3. In phase 1, we add processors for the current cluster. It is guaranteed that adding processors will decrease the T_{comp} component of T_c. The algorithm computes two things in *get_best_config* — the best processor configuration based on the previous configuration and the current cluster, and the *partition_map*. It has the property that once P_j is computed for cluster C_j, it is not modified as additional clusters are considered. Thus, phase 1 is a greedy algorithm. For each cluster considered it locates the best number of processors by a binary search procedure similar to the method described for cluster ordering. The difference is that here we are looking for the minimum T_c for the current cluster C_i assuming a fixed number of processors already selected for the previous clusters. The best configuration is stored during this initial phase. The worst-case complexity of phase 1 is also θ ($mlogP_{max}$).

The addition of processors will never decrease T_{comm}, though it may remain unchanged. In phase 2, we try to reduce the T_{comm} component of T_c. The total communication cost is a function of the communication cost contributed by each cluster (Eq.3.7). The cluster that contributes the maximum communication cost is targeted for reducing the overall communication cost. In phase 2, we add processors for the current cluster while removing processors from the cluster that contributed the largest communication cost.

```
Order clusters C1 .. Cm by Tc
Initialize curr_config, min_cost
For each cluster Ci {
    // Phase 1 -- Try to reduce T_comp
    // Determine config that yields min Tc given previous Pj (j<i)
    best_curr_config = get_best_config (curr_config, Ci);
    // min_cost is stored

    // Phase 2 -- Try to reduce T_comm
    curr_config.Pi = 0;
    min_phase2 = MAXFLOAT;
    // Repeatedly trade processors in Ci with processors in Ck (k<i)
    // where Ck is the cluster with the largest communication cost
    // Ck may change during phase 2 -- if it is the current cluster, exit
    while ((curr_config.Pi <= Vi) && (k!=i)) {
        curr_config.Pi++;
        curr_config.Pk--;
        Tc = get_Tc (curr_config);
        if (Tc < min_cost) {
          best_curr_config = curr_config;
          min_cost = Tc;
        }
        // Optimization: if Tc increases in phase 2 then exit phase 2
        if (Tc < min_phase2)
          min_phase2 = Tc;
        else break;
    }
    curr_config = best_curr_config;
}
return best_curr_config;
```

Figure 4.3: Pseudo code for Heuristic H_1

Removing processors from a cluster has the effect of reducing the communication cost contributed by that cluster by reducing the contention for communication resources. The idea is that additional communication bandwidth may be made available by reducing the processors in one cluster and increasing the processors in another.

This technique is guaranteed to reduce T_{comm}, but the impact on T_c is unpredictable since T_{comp} may increase since we are trading potentially faster processors for slower ones. The cluster that contributes the largest communication cost may change during the course of phase 2 as processors are traded. The configuration that yields the minimum T_c

after both phase 1 and phase 2 is stored. This is the starting configuration that is used as the next cluster is considered. H_1 does not terminate until all clusters are explored. The worst-case complexity of phase 2 is θ (mP_{max}). This worst-case is only a concern for small problems that would not be able to amortize this overhead. But phase 2 will terminate if T_c increases during this phase. In practice the average complexity is much smaller than the worst-case. Furthermore, there is a practical limit on how large P_{max} will be based on the number of processors within a parallel machine, or the number of stations allowed on an ethernet segment or FDDI ring. We expect *m* to be small (less than 50) in local-area metasystems. For wider-area metasystems, a strategy that limits the number of clusters under consideration will be needed.

In Chapters 6 and 7, we present simulation and experimental results that show H_1 is a feasible algorithm. The results indicate that performance within 10% of optimal is obtained over 90% of the time in simulation. Experimental results also yield excellent performance. The observed worst-case deviation from optimal was around 40% in simulation (and this was quite rare), but a more rigorous analysis of a worst-case bound is the subject of future work. In simulation we have observed that H_1 rarely falls into local minima. The reason is cluster ordering and the phase 2 stage of the algorithm. Cluster ordering is an effective strategy for resource selection and phase 2 explores the processor configuration space in a non-greedy fashion increasing the likelihood that local minima will be avoided. We observed in simulation that a random cluster ordering causes the method to fall into local minima by selecting less effective processors for the problem. Phase 2 is needed to the avoid the local minima that may occur due to a *max* in T_c.

Heuristic H_2

Heuristic H_2 is a special case of H_1 that is suited to workstation network environments in which communication capacities are the same within each cluster in the metasystem (e.g., ethernet-based clusters only), and routing costs are high. H_2 exploits features of

this environment to simplify the processor selection algorithm and has smaller overhead than H_1.

The algorithm begins by ordering the clusters as in H_1. The next stage of the algorithm explores the processor configuration space in a greedy fashion much like phase 1 for H_1 with an θ ($mlogP_{max}$) worst-case complexity. All processors of a cluster are selected before processors in the next cluster are considered thus avoiding router crossings if possible. This algorithm tries to maintain communication locality by avoiding the router penalty and potential data conversion overhead. The algorithm terminates when adding processors in the current cluster causes T_c to increase, and is sketched in Figure 4.3.

```
Order clusters C1 .. Cm by Tc
Initialize curr_config, min_cost
For each cluster Ci {
      / / Determine config that yields min Tc given previous Pj (j<i)
      best_curr_config = get_best_config (curr_config, Ci);

      / / If Tc has increased we are done
      if (best_curr_config.cost > min_cost)
          break;
      else {
          curr_config = best_curr_config;
          min_cost = best_curr_config.cost;
      }
}
return best_curr_config;
```

Figure 4.4: Pseudo code for Heuristic H_2

The worst-case order of this algorithm is θ ($mlogP_{max}$). H_2 differs from H_1 in that it uses a simpler strategy for exploring the configuration space. In practice it will also be more efficient due to the greedy termination condition. H_2 was the precursor for H_1 and experimental results for H_2 were published in [93]. Some preliminary results for H_1 were published in [91]. The performance results for a homogeneous network of Sun workstations and an Intel Paragon indicated that completion times close to the minimum were achievable for real data parallel computations.

This dissertation has focused on the more general heuristic H_1 and we drop the name H_1 in the remainder of the thesis. We refer to H_1 as the partitioning method in subsequent chapters.

4.2 Task Placement

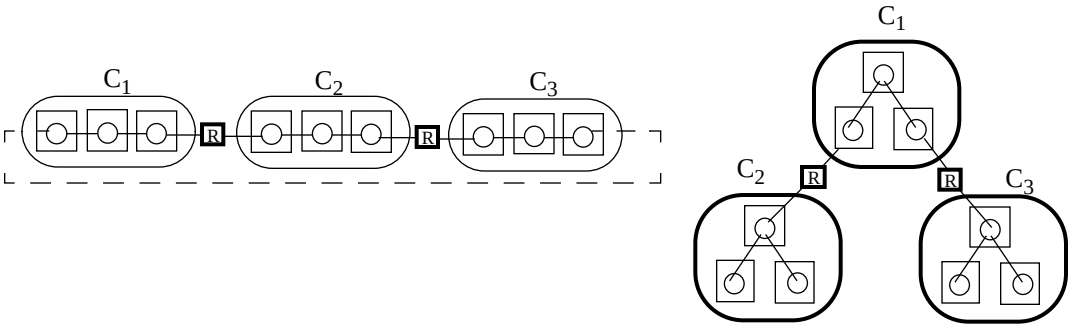
Placement is the assignment of tasks to processors and has two principle objectives, ensuring one task per processor, and assigning tasks in a manner that reduces communication cost. Assigning one task per processor is needed to achieve processor load balance for SPMD computations. Placement uses a form of *co-scheduling* to collectively assign tasks to specific processors to guarantee one task per processor. The second objective of placement is more difficult and is the subject of this section.

Assigning tasks in a communication-efficient manner must rely on information about the communication and interconnection topologies. Reducing communication costs is achieved by (1) maintaining communication locality (i.e., avoiding router crossings and potential conversion) and (2) effectively exploiting communication bandwidth within clusters. The former is achieved by *inter-cluster* placement and the objective is to minimize communication costs between clusters. Empirical evidence suggests that this is a large source of overhead. The latter is achieved by *intra-cluster* placement and the objective is to minimize communication costs within clusters. Intra-cluster placement is also known as *mapping* or *embedding* and has been widely studied [7][48][54][70]. Both inter- and intra-cluster placement exploit available topology information and the regular nature of the communication topology. For both stages of placement information about the dominant communication topology is used.

4.2.1 Inter-cluster Placement

Inter-cluster placement uses communication topology information to minimize the amount of communication that crosses the router. We have developed inter-cluster placement strategies for the prototype application topologies: *1-D*, *ring*, *tree*, and *broadcast*. In

Figure 4.5 the *1-D*/*ring* (the ring is indicated by the wrap-around arc) and *tree* topology are decomposed across three processor clusters — the tasks are the circles and the squares are the processors. Observe that the amount of communication that crosses the router is minimized. Two messages cross the router each cycle between each pair of communicating clusters. These inter-cluster placement strategies are topology-preserving in the sense that each group of tasks assigned to a cluster maintains the topology. For example, each group of tasks assigned to C_1, C_2, and C_3 in the *1-D* topology each communicate in a *1-D* topology. Similarly for the *tree* topology. The *broadcast* topology does not exhibit locality but a strategy that assigns the master to the cluster that contains the largest number of tasks reduces the amount of router communication.



a) *1-D* and *ring* b) *tree*

Figure 4.5: Inter-cluster placement

Inter-cluster placement depends on cluster ordering. In processor selection we have determined how many processors a cluster will contribute. This is the same as the number of tasks assigned to each cluster since each task is assigned to one processor. Cluster ordering governs the assignment of tasks to clusters. For example in Figure 4.5 we show the task assignment for the cluster order C_1, C_2, C_3. Tasks in the *1-D*/*ring* topology are assigned left-to-right to C_1, C_2, C_3. In the *tree* topology the root task and its subtree are assigned to C_1, and the subtrees corresponding to the other tasks are assigned in the order C_2, C_3 in an attempt to minimize tree height. We are trying to strike a balance between reducing tree height and minimizing concurrent router crossings for the tree. Simply try-

ing to reduce router communication may result in a tree of greater height which will lead to a larger communication overhead. These procedures generalize to any number of clusters.

These inter-cluster placement strategies determine the total communication cost T_{comm} [τ]. In the current implementation we assume that the routing costs between any group of clusters is the same — a reasonable assumption for local-area environments. For wide-area environments, non-uniform routing costs and inter-cluster network topology information will be needed for inter-cluster placement.

4.2.2 Intra-cluster Placement

Intra-cluster placement assigns tasks to specific processors within a cluster. Intra-cluster placement depends both on the communication topology and the interconnection topology. Two factors that contribute to intra-cluster communication costs are dilation and contention. Dilation is the number of communication hops. High dilation and contention will tend to limit the exploitable communication bandwidth. Intra-cluster placement should keep the average dilation small and limit contention. For example, a grey-scale mapping of a *1-D* topology onto a hypercube achieves minimal dilation and contention [48]. On the other hand, a random placement suffices on a shared bus interconnect for any communication topology. For multicomputers the embedding may also depend on the dimension of the mesh partition or sub-cube. There is a rich literature on the mapping problem and many of the algorithms are well known [48][54][70].

Traditionally mapping algorithms have been applied within a static compile-time scheduling framework. We use these algorithms to make runtime placement decisions. A subset of these algorithms have been implemented and made available for use at runtime. We have implemented intra-cluster mapping strategies for the workstation environment only. The cluster communication cost functions T_{comm} [C_i, τ] are benchmarked using these

intra-cluster placement strategies. This guarantees that the cost prediction that ultimately guides the partitioning and placement stages will be accurate.

Many problems have multiple communication topologies. For example, a *1-D* topology might be used for a nearest-neighbor communication and a *tree* for a global communication. Our strategy is to perform intra-cluster placement for the dominant topology first followed by intra-cluster placement for the other topologies. The current implementation can support problems containing *both 1-D* and *tree* topologies.

An interesting case is the *2-D* topology. In a homogeneous environment, mappings for the static *2-D* topology have been developed. In the heterogeneous environment, it may not be possible to preserve the *2-D* topology since processors of different types may be assigned different size regions of the data domain and the communication topology becomes irregular, see Figure 4.6. The processor that is assigned the shaded region will need to communicate with 5 processors. A strategy for dealing with the *2-D* topology is the subject of future work.

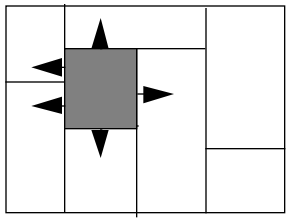


Figure 4.6: 2-D problem

We also handle the *hybrid-tree* topology discussed in Section 3.2.4. Recall that in this topology, the leaves perform the computation while the interior nodes perform communication. Intra-cluster placement first insures that the leaves are placed one per processor for load balance. The interior nodes are light-weight and may be placed several to a processor. The placement method first tries to place them on idle processors and then tries to ensure that each processor has roughly the same number of tasks that correspond to interior nodes, also for load balance.

In this chapter we have presented several promising heuristics for the partitioning and placement problem. Both greedy and non-greedy algorithms were described. Partitioning and placement were performed using a set of runtime cost functions for computation and communication that have been constructed from system resource and program information.

Chapter 5 Implementation

This chapter presents an implementation of the scheduling framework in the Mentat-Legion[1] parallel processing system. We have completed the implementation for heterogeneous workstation networks. The heart of the scheduling framework is Prophet — a PaRtitiOner for Parallel programs in a HETerogeneous environment. We describe Mentat and Legion and all components of the Prophet-Legion implementation including the callback and program interface, system configuration, and resource availability.

5.1 Prophet

Prophet implements the middle stage of the scheduling framework, partitioning and

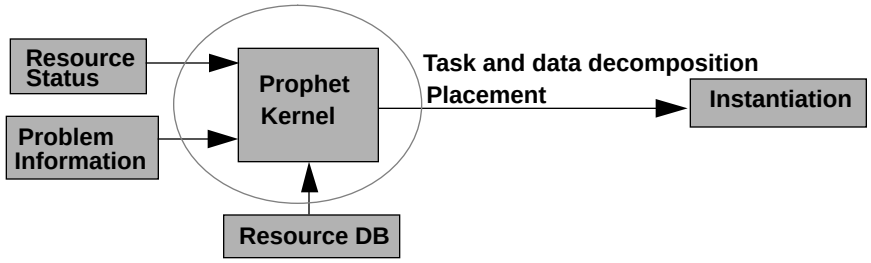


Figure 5.1: Prophet

placement, and defines a set of interfaces, see Figure 5.1. The core of Prophet is a runtime

1. Mentat-Legion refers to a transitional stage between the Mentat and Legion parallel processing systems.

kernel that can be integrated into a number of other parallel processing systems that support the Prophet interface and satisfy a number of system requirements.

A primary requirement is that the host parallel processing system must be able to support our heterogeneous network model and some form of resource or configuration database as described in Chapter 3. This is needed to implement resource availability, the first stage of the framework. Another requirement is that the host system provide some form of callback mechanism to make program information available. All of these requirements are needed by the Prophet kernel to support partitioning and placement.

In addition there are three requirements for instantiation — a data format conversion capability, a dynamic worker or task creation capability, and a mechanism to insure that binaries for the worker task are available for each architecture type and resident on the appropriate file system. Support for dynamic task creation depends on what the underlying operating system provides. Data format conversion may be implemented within the host communication system. The host communication system is also assumed to support message-passing between all machines in the environment.

We present an integration of Prophet into the Mentat-Legion parallel processing system and describe how each piece in the picture of Figure 5.1 is implemented. The current Prophet implementation consists of approximately 2000 lines of C++ code.

5.2 Legion

Legion is a distributed parallel processing system based on Mentat. Legion will provide a set of services that enables wide-area parallel and distributed computing [34]. As in Mentat, Legion programs are collections of communicating objects. One of the primary objectives of Legion is to provide a seamless virtual computer that hides much of the complexity inherent in managing a distributed collection of resources. Seamless parallel processing in Legion means that the system must be able to locate processing resources and make scheduling decisions automatically for the user. The integration of Prophet into

Legion is aimed at providing this capability for data parallel computations.

5.3 Mentat-Legion Implementation

Prophet Kernel

The Prophet kernel is responsible for making partitioning and placement decisions based on problem and resource information. Problem information is provided by a callback interface and resource information by a resource database and resource status interface. The kernel implements the algorithms for partitioning and placement discussed in Chapter 4. Partitioning and placement information are computed and stored in a set of data structures that are made available by a Prophet kernel call. The current implementation of the Prophet kernel is written in C++ and is compiled with the Mentat-Legion runtime system library also written in C++. All application code including the worker implementation link this library.

The kernel also manages the set of workers or tasks created by instantiation. It treats the set of workers as a collection, and defines two useful variables that the worker implementation can use: `COLLECTION_ID`, the id of the worker, and `NUM_COLLECTION`, the number of workers in the collection. In the Mentat-Legion implementation, this id maps into the Mentat object name, which is needed to enable communication between workers. The current implementation supports a number of application communication topologies, *1-D*, *ring*, *tree*, *hybrid-tree*, *broadcast*, *RPC*, and *other*, and the following operations are supported on collections of these types, see Figure 5.2.

1-D : `NORTH(), SOUTH()`
ring: `PRED(), SUCC()`
tree, *hybrid-tree*: `LCHILD(), RCHILD(), PARENT()`
hybrid-tree: `LEAF()`
broadcast: `MASTER(), SLAVE(k)`

Figure 5.2: Collection operations

These functions may be called by the workers to determine their communicating partners

based on topology and return the name of the communicating worker, e.g., the name of a worker's north sibling. For *broadcast* the master can obtain the name of the k^{th} slave worker. The function `LEAF()` is useful for the *hybrid_tree* topology — it returns true if the calling worker is a leaf, otherwise false. *RPC* is a point-to-point communication between two objects and is useful in the Mentat-Legion implementation. The purpose of the remaining functions is straightforward. The topology *other* refers to any unimplemented topology.

Configuration

The heterogeneous network model is easily implemented in Mentat-Legion which already defines a notion of cluster. We restrict the Mentat-Legion cluster to include only homogeneous processors. The configuration information is stored in a database that we have extended to support Prophet. The database is encapsulated by a C++ class `configdb`. In Figure 5.3 we present a specification for a configuration containing a Sun Sparc2 cluster with 8 processors and SGI cluster with 6 processors, both on ethernet. This specification corresponds to the information in Figure 3.2. The communication functions are specified by values for the constants c_1, c_2, c_3, and c_4 respectively in (Eq.3.3), and f is assumed to be linear. In the workstation environment, f will be linear. In a true metasystem environment, specification of a non-linear f will need to be supported in the future. For the routing functions the values refer to the constants r_1 and r_2 in (Eq.3.1). For the conversion functions the value refers to e_1 in (Eq.3.2). This specification will need to reflect different types of possible conversions in the future. These values allow Prophet to construct the appropriate communication cost functions for workstation networks. RPC is for a 0-byte message. The peak communication bandwidth is COMM_BANDWIDTH and latency is determined to be the one-way RPC latency, so there is no need for an additional specification. The communication parameters were determined by benchmarking a set of communication programs written using the Mentat-Legion communication system MMPS [38].

```
CLUSTER SPARC2s
{
cluster01.cs.Virginia.EDU
cluster02.cs.Virginia.EDU
...
cluster08.cs.Virginia.EDU
CLUSTER_TYPE SUN4
TOPOLOGY BUS
MANAGER cluster01.cs.Virginia.EDU
FLOPS 170
MIPS 170
COMM_BANDWIDTH 10.0 // Mbit/sec
// All in msec
RPC 4.1
BCAST .9 2.1 .003 .00116
...
TREE .5 2.1 .00051 .0019
TOP_DEFN CHORDAL .1 .1 .1 .1
TOP_DEFN IRREG_TOP1 LOCAL
TOP_DEFN IRREG_TOP2 GLOBAL
}

ROUTER SGIs SPARC2s 1.2 .00008
CONVERSION SGIs SPARC2s 0.0
```

```
CLUSTER SGIs
{
sgi-1.unixlab.Virginia.EDU
sgi-2.unixlab.Virginia.EDU
...
sgi-6.unixlab.Virginia.EDU
CLUSTER_TYPE SGI
TOPOLOGY BUS
MANAGER sgi-2.cs.Virginia.EDU
FLOPS 360
MIPS 360
COMM_BANDWIDTH 10.0 // Mbit/sec
// All in msec
RPC 3.6
BCAST .4 2.0 .000073 .00145
...
TREE .7 1.8 .00012 .0014
}
```

Figure 5.3: Example configuration

We also propose a mechanism for user-defined topologies that is illustrated above. The topology is given a name (e.g., CHORDAL) and a set of cost coefficients if they are known. If the cost parameters are unknown then the user may specify whether the topology is LOCAL or GLOBAL. If the topology is LOCAL then the system will use the *1-D* cost function as an approximation or the *broadcast* cost function if it is GLOBAL. A method for specifying placement information including whether the topology is a SAT or CAT will be needed for user-specified topologies. User-specified topologies are currently unimplemented but we have provided a generic topology called *other* that Prophet defines conservatively — it is assumed to be a SAT with linear *f*, placement is random, and the cost coefficients are formed as an average of the cost coefficients of the other specified topologies. If cost functions are omitted for this or any other topology then the optimistic cost

function of (Eq.3.4) can be used. This default is currently unimplemented.

The Mentat-Legion system runs a daemon process known as the *instantiation manager* on each host in the configuration that is responsible for collecting load information. One *instantiation manager* per cluster is designated as the *manager* in our model.

Resource Availability

The current implementation of resource availability is based on a sender-initiated probe of all hosts in the local-area configuration to obtain their load status. When a scheduling request arrives, load and availability information is determined and an aggregate of the information is returned. The manager mechanism is not yet fully implemented since a complete implementation of resource availability is outside the scope of this dissertation. At present each host redundantly stores a copy of the resource database as well. Once the manager mechanism is in place, a more scalable load collection strategy based on the picture of Figure 3.1 can be implemented.

The current implementation considers a processor below a run-queue-length load threshold of .33 to be available. This guarantees that at least 75% of the CPU will be available for the data parallel problem at the time the computation begins. Processors are ordered by their load value within each cluster. We choose processors with the lightest load first before processors with a larger load are considered.We have put hooks into the Prophet kernel to use load information for adjusting the cost functions in the future. Research is needed to be able to quantify the impact of load on the computation and communication costs. A related problem is the need for dynamic load balance. These topics are discussed in Chapter 8.

Callback Interface

We have implemented a C++ callback interface by defining an abstract base class `domain` see Figure 5.4. The callbacks are member functions on this class. The description of these callbacks was provided in Chapter 3.

```
class domain {
char** PV;
public:
      virtual domain (char** curr_PV);
      virtual phase dominant_comp_phase (int np)=0;
      virtual phase dominant_comm_phase (int np)=0;
      virtual phase_rec num_phases ()=0;

      virtual comp_rec comp_complexity (int np, phase comp_phase)=0;
      virtual int numPDUs (phase comp_phase)=0;
      virtual cost_rec arch_cost (host_types proc, phase comm_phase)=0;

      virtual comm_rec comm_complexity (int np, phase comm_phase)=0;
      virtual phase overlap (phase comp_phase) = 0;
      virtual top topology (phase comm_phase)=0;
};
```

```
struct cost_rec {
      float PDU_cost;
      float non_PDU_cost;
};

struct phase_rec {
  int comp_phases;
  int comm_phases;
};
typedef enum {1D, ring, tree} top;
```

```
struct comp_rec {
      float PDU_inst;
      float non_PDU_inst;
};

struct comm_rec {
      int PDU_bytes;
      int non_PDU_bytes;
};
typedef int phase;
```

Figure 5.4: Callback interface

There are several callbacks added to the group described in Chapter 3:

`dominant_comp_phase()` returns the dominant computation phase
`dominant_comm_phase()` returns the dominant communication phase
`num_phases()` returns the number of phases

These callbacks are needed since it is possible that the dominant phases depend on problem parameters known at runtime. The structure PV is a parameter vector and is similar to argv. It may contain any number of problem parameter values needed to implement the callbacks. Currently the programmer marshals the problem parameters into PV and instantiates the domain with this vector as an argument. An example is provided in the next section. Notice that the number of processors, *np*, is passed as a parameter to the callbacks since some callbacks may depend on it. The phases are represented as integers and it is up to the

implementation of the domain class to map these integers into the program phases. The phases are assumed to be numbered from 0 to `num_phases()` - 1. With appropriate language support this mapping could be managed by a compiler.

The domain class must be derived and implemented for a particular data parallel computation. For example, we have defined a domain class for stencil, `stencil_domain`, that provides information about the *1-D* stencil computation described in Section 3.2. In Figure 5.5 we show the implementation of the `comp_complexity()` callback for the stencil problem. This problem has one computation phase, hence the simple switch statement. This callback depends on a single problem parameter, the problem size *N* that is extracted from the parameter vector PV.

Callback specification can be a tedious task for the programmer. One solution is to provide libraries of callbacks for well-known computational structures such as stencil problems. The programmer would extend these classes by derivation and not have to reimplement the entire domain class from scratch. A more attractive idea is to have the compiler generate the callbacks. As was discussed this is unlikely to be a general solution for irregular problems but may have promise for regular problems.

```
class stencil_domain : domain {
public:

   comp_rec comp_complexity (int np, phase comp_phase) {
      int N = atoi (PV[0]); // extract problem size
      comp_rec CR;
      switch (comp_phase) {
        case 0 :
          CR.PDU_inst = 5*N; // 5 fp operations per PDU in this problem
          CR.non_PDU_inst = 0;
          break;
      }
      return CR;
   }

    ...
```

Figure 5.5: Implementation of stencil callbacks

Program Interface

The program interface to the Prophet kernel is provided by a function `partition` that returns the partition and placement information in a set of data structures. We have provided a Mentat-Legion facility to support instantiation, `DP_create`, that instantiates a Mentat object (i.e., a worker) on each selected processor, and communicates the list of workers to each worker. This facility allows the worker implementation to establish the communication topology and to determine its communicating partners via the functions in Figure 5.1.

In Figure 5.6 we present a partial main program for *1-D* stencil written in MPL. The

```
main() {
     partition_rec *PR;
     stencil_worker *workers, mo;
     stencil_domain *dom;
     DD_floatarray *Grid;
     ...
     // Problem-specific code: (N, Grid, iters are read from file)
     PV[0]= itoa (N); // marshal PV for problem instance
     dom = new stencil_domain (PV); // instantiate domain

     PR =  partition (dom);
     mclass* workers = (stencil_worker*) DP_create (PR, mo);

     // Application-specific code
     1D_grid = 1D_carve (Grid, PR.partition_map);
     for (int i=0; i<PR.total; i++)
         workers[i].init_grid (1D_grid[i], N);
     for (int j=1; j<=iters; j++)
         for (int k=0; k<PR.total; k++)
           workers[k].compute_grid ();
     ...
}
```

Figure 5.6: Stencil main program

main program begins by constructing PV and instantiating `stencil_domain`. The `stencil_domain` object is then passed to `partition` — this will enable Prophet to invoke the callbacks. A call to `DP_create` is then made to place a Mentat object on each processor based on the information contained in PR. We omit the definition of

`partition_rec` since it is a fairly complex structure. The workers are instances of the Mentat class `sten_worker` — the definition of `sten_worker` and an example member function is given in Figure 5.7. `DP_create` returns the list of created Mentat objects. We have decoupled `partition` and `DP_create` since `partition` is a generic kernel call while `DP_create` is a Mentat-Legion specific call.

The information returned by `partition` is also needed for data decomposition. In this problem, the *partition_map* is used to decompose the grid into 1-D chunks via the call to `1D_carve`. The implementation of stencil relies on facilities in a library that manages 1-D and 2-D data structures known as `DD_array`. The grid is represented as a memory-contiguous 2-D float array, `DD_floatarray`. The implementation of `1D_carve` uses library facilities to extract the appropriate pieces of the grid.

The `sten_worker` stores it's portion of the problem in a set of member variables and defines a number of member functions — `init_grid` initializes each worker with it's piece of the problem, `compute_grid` initiates a worker to begin the stencil computation, and `put_top/bot` communicate a border row to neighboring workers. In the implementation of `compute_grid`, the neighboring workers are determined by calls to `NORTH()` and `SOUTH()` and the stencil operation is performed for a fixed number of iterations. We omit the code for `update_grid`, the function that performs the five-point stencil on the stored rows of the subgrid.

In this chapter we have described the Mentat-Legion implementation of the scheduling framework for workstation networks. The implementation includes the program interface, resource availability, and the Prophet kernel. The latter implements the algorithms for partitioning and placement that automate scheduling.

```
persistent mentat class sten_worker {
      float *top, *bot;              // cushion rows from communicating workers
      DD_floatarray *subgrid;        // worker portion of subgrid
      int dim;                       // columns in grid
public:
      void init_grid (DD_floatarray* sgrid, int N);
      void compute_grid ();

private:
      // Communication functions
      void put_bot (DD_floatarray* row);
      void put_top (DD_floatarray* row);

      void update_grid (int num_rows);
};

void sten_worker::compute_grid () {
      int num_rows = subgrid->num_row();
      sten_worker *north, *south;
      DD_floatarray *mytop, *mybot;

      // Get neighbors
      north = (sten_worker*) NORTH();
      south = (sten_worker*) SOUTH();

      // Extract borders and communicate to neighbors (if any)
      mytop = subgrid->extract_region (0, 0, 0, dim-1);
      mybot = subgrid->extract_region (num_rows-1, 0, num_rows-1, dim-1);
      if (north != 0)
           north->put_bot (mytop);
      if (south != 0)
          south->put_top (mybot);
      ...
      // compute on the subgrid -- update_grid computes 5-pt stencil for each row
      update_grid (num_rows);
      rtf (0);
      return;
}
```

Figure 5.7: Sten_worker implementation

Chapter 6 Simulation Study

This chapter presents the results of two simulation studies — a performance study of the partitioning method, and a study into wide-area parallel processing. A simulator for Prophet called *Prophesy* has been developed to perform the simulation studies. We have performed a simulation study into the expected performance of the partitioning method to show that it has applicability to a variety of problem types in different metasystems. The performance results indicate that the partitioning method has excellent average-case behavior over a wide range of problem granularities and application communication topologies. These results complement the experimental results in Chapter 7. The wide-area parallel processing study provides some insight into the granularity requirements for wide-area parallel processing. We discuss each study in turn.

6.1 Prophesy

Prophesy is a simulation system for Prophet that supports the simulation of synthetic metasystems and synthetic data parallel problems. It contains 1400 lines of C++ code. Prophesy is built using the Prophet kernel, but replaces the resource and program inputs (circled boxes) with synthetically generated information, see Figure 6.1. Prophesy also computes the optimal solution and generates a comparison with the heuristic solution.

To study the performance of the partitioning method, we simulated the partitioning

of data parallel computations in three classes of metasystems, M1, M2 and M3. The objective is to determine how close the partitioning heuristic comes within optimal on average. M1 contains a collection of hosts and assumes an unequal communication capacity among clusters. An example of M1 would be a cluster of ethernet-connected workstations connected to a cluster of FDDI-connected workstations. M2 is a mixed environment of single

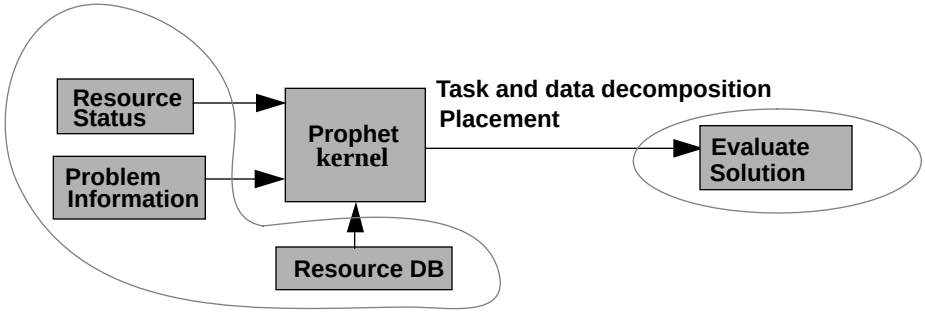


Figure 6.1: Prophesy

CPU hosts and mesh-connected multicomputers and assumes an unequal communication capacity among clusters. An example of M2 would be a cluster of ethernet-connected workstations and an Intel Paragon. M3 contains a collection of hosts and assumes an equal communication capacity among clusters. An example of M3 would be clusters of workstations all connected by ethernet. These environments differ in the form of the communication cost functions.

A metasystem environment is determined by generating processor clusters and the information described in Figure 6.2. All generated parameters are uniformly distributed over a fixed range. The ranges are limited to values that have been empirically observed or have been published elsewhere. For example, a latency constant is restricted to be in the millisecond range on an ethernet-based cluster, while a bandwidth constant is restricted to be in the microsecond range. The latency and bandwidth constants apply to parameters for the communication, router, and conversion cost functions. The value ranges for the router cost functions are based on a local-area environment. The router penalty increases the

metasystem parameters
num_clusters = [1 .. 5]
num_processors_per_cluster = [1 .. 10]
processor_rate = [1 .. 100] mflops
interconnect = [mesh, bus]
latency_constant = [0 .. 1] msec
bandwidth_constant = [.1 .. 10] μsec/byte
conversion_constant = [0 .. 1] μsec/byte

Figure 6.2: Simulation parameters (environments)

latency and degrades the bandwidth for wider-area configurations. The number of clusters and processors also reflect reasonable values for a local-area environment. Due to the length of the simulation runs we have opted for a modest cluster size though the results obtained for a few larger systems (on the order of 100 processors per cluster) are in agreement with the results we present.

We simulated applications with the following communication topologies: *ring*, *1-D*, and *tree*. For each topology, the communication cost functions are determined by generating the cost constants in (Eq.3.3), i.e., c_1, c_2, c_3 and c_4. For bus interconnects, f is linear in the number of processors for all topologies. For the mesh-based multicomputer, f is $log(p)$ for the *tree* topology, and nearly-independent[1] of p for the *1-D* and *ring* topology due to a dilation one embedding (i.e., intra-cluster placement) of the *1-D* and *ring* topology onto the mesh. The total communication cost T_{comm} is computed by the functions in (Eq.3.7).

A problem instance is determined by generating the callback information specified in Chapter 3. All generated parameters are uniformly distributed over a fixed range, see Figure 6.3. The problem instance contains a communication phase and a computation phase. A problem instance may have overlap between these two phases. The values for *comp_complexity* and *comm_complexity* are generated to simulate a range of problem granularities. To keep things simple, the *PDU* independent term for *comp_complexity* is 0

1. This is achieved by setting cost constants c_2 and c_4 very small.

and the *PDU* dependent term for *comm_complexity* is 0. That is, the amount of time spent in computation depends only on the number of *PDU*s assigned to each worker, and the amount of data communicated is independent of the number of *PDU*s assigned to a worker. These restrictions may be easily relaxed.

problem parameters
top = [tree, ring, 1-D]
NumPDUs = {1, 100, 500, 1000, 5000, 10000}
comm_complexity = [1 .. NumPDUs] bytes
arch_cost = [.01 .. 1] μsec/instruction
comp_complexity = [1 .. 10000] instructions
overlap = [yes, no]

Figure 6.3: Simulation parameters (problems)

For each problem instance, a number of values for *comm_complexity* on the interval [1 .. *numPDUs*] are simulated. In real codes the message size normally depends on the how the problem was decomposed. We simulate a problem size, *numPDUs,* for the following the values: 1, 100, 500, 1000, 5000, 10000. The *arch_cost* is inversely proportional to the peak processor rate.

6.2 Performance of Partitioning Method

We have applied Prophesy to a range of synthetic problem instances and metasystem configurations. We simulate the partitioning of each problem instance in each metasystem. Prophesy measures the predicted elapsed time achieved by the heuristic and compares this to the elapsed time for optimal partitioning which is obtained by an exhaustive search of the processor configuration space. We simulated 50 metasystem environments in each class (M1, M2, M3) and simulated 50 problem instances for each metasystem. For each problem instance, we simulate the 6 problem sizes for *NumPDU*s listed in Figure 6.3 and 3 message sizes for *comm_complexity* for a total of 18 runs per problem instance. The total number of problems simulated is 900 (50 x 18) and the total number of runs is 45,000 (50 metasystems x 900) per *experiment*. The data for each experiment presented in the subse-

quent tables is the average of 45,000 runs. Each problem instance contains a computation and communication phase with a *ring*, *1-D*, or *tree* topology that may or may not be overlapped with the computation phase. The use of overlap and different communication topologies change the form of the objective function T_c. Since the performance of the partitioning method depends on the nature of T_c, simulation is a viable way to study the performance of the method.

Prophesy computes a processor configuration and data domain decomposition for the given problem instance in the synthetic metasystem environment. The quality of the results are determined by computing T_c for this configuration using the Prophet kernel, $T_c^{Prophet}$. We compare this value to the value for the optimal T_c, $T_c^{optimal}$. The optimal is obtained by determining the processor configuration and data domain decomposition that produces a minimum value for T_c. Under the assumptions detailed in Section 4.1.2, the processor configuration and data domain decomposition that produces a minimum T_c will also produce a minimum total elapsed time. The experimental results given in Chapter 7 confirm that T_c is an excellent predictor for total elapsed time. For a selected processor configuration, an optimal data domain decomposition is one in which the processors are load balanced under the assumption of synchronous communication. Consequently, the optimal solution need not explore all possible assignments of *PDU*s to processors. Instead, the optimal solution is determined by an exhaustive search of the processor configuration space with the data domain decomposition computed for each configuration by (Eq.4.5).

The optimal solution also considers all possible assignments of tasks to processor clusters (inter-cluster placement). The best intra-cluster placement is assumed to be provided by the generated T_{comm}. That is, the synthetic coefficients generated for T_{comm} are assumed to reflect the best intra-cluster placement strategy.

The simulation results are validated by the experimental results in Chapter 7. In Chapter 7 we show that the predicted value of T_c that guides the partitioning method agrees with the observed value for T_c for a suite of real data parallel codes and that T_c is an excel-

lent predictor for total elapsed time, $T_{elapsed}$.

The simulation results are first broken down by problem type, overlapped and non-overlapped. Next the results are divided by metasystem type (M1, M2, or M3). Within each metasystem type, the results are further broken down by communication topology. We also simulated environments that required routing and conversion and those that did not. For all experiments we ran the partitioning method with and without cluster ordering. Although cluster ordering adds overhead, it improves the performance of the partitioning method significantly. We show the performance with and without cluster ordering to highlight the importance of this scheme.

In this first set of experiments we consider problems that do not have computation overlapped with communication. We present the percentage of experiments that were within 5% and 10% of optimal respectively, see Table 6.1-Table 6.3. We consider an elapsed time for T_c within 10% of optimal to be acceptably good performance. The results indicate that this is achieved over 90% of the time. Each value in the table is the average of 45,000 distinct runs.

topology	**with cluster ordering % of optimal**		**without cluster ordering % of optimal**	
	5%	10%	5%	10%
ring	98.6	99.5	63.9	70.7
1-D	89.3	94.4	72.7	81.2
tree	91.6	95.3	61.5	68.8

a) No router/conversion

topology	**with cluster ordering % of optimal**		**without cluster ordering % of optimal**	
	5%	10%	5%	10%
ring	98.7	99.6	61.4	67.7
1-D	88.9	94.6	63.9	70.7
tree	92.6	95.9	52.5	59.7

b) router/conversion

Table 6.1: Simulation results for M1, hosts with unequal communication capacity. Table a) contains the simulation of metasystems with router and conversion costs included. Table b) does not simulate these costs.

It is also important to point out that 100% of the runs were within 40% of optimal. That is, the worst performance we observed was 40% greater than optimal and this occurred very rarely. We also see that the inclusion of router and conversion overhead does not significantly perturb the performance of the algorithm. This validates our cluster ordering

topology	with cluster ordering % of optimal		without cluster ordering % of optimal	
	5%	10%	5%	10%
ring	97.7	99.3	67.5	76.4
1-D	91.4	95.0	69.9	76.5
tree	89.2	91.7	63.3	70.2

a) No router/conversion

topology	with cluster ordering % of optimal		without cluster ordering % of optimal	
	5%	10%	5%	10%
ring	98.8	99.7	64.3	71.1
1-D	92.3	96.4	65.9	73.0
tree	88.1	91.6	63.5	71.1

b) router/conversion

Table 6.2: Simulation results for M2, workstations and multicomputers. Table a) contains the simulation of metasystems with router and conversion costs included. Table b) does not simulate these costs.

topology	with cluster ordering % of optimal		without cluster ordering % of optimal	
	5%	10%	5%	10%
ring	98.6	98.7	59.2	66.8
1-D	94.4	98.4	59.1	66.7
tree	92.8	96.2	58.9	63.8

a) No router/conversion

topology	with cluster ordering % of optimal		without cluster ordering % of optimal	
	5%	10%	5%	10%
ring	98.8	99.6	54.5	59.5
1-D	92.7	97.6	54.0	61.8
tree	93.1	96.8	58.9	63.8

b) router/conversion

Table 6.3: Simulation results for M3, hosts with equal communication capacity. Table a) contains the simulation of metasystems with router or conversion costs included. Table b) does not simulate these costs.

strategy based on T_c for local-area metasystems. Cluster ordering considers the clusters in isolation and includes only communication costs within the cluster, ignoring router and conversion costs between clusters. The benefit obtained by the use of cluster ordering is substantial, a 40% improvement — from 69% to 97% (fall within 10% of optimal) approximately. The performance results differ slightly between the different topologies with performance higher for the *ring* than for either the *1-D* or *tree* topologies. The reason is that T_c is a more complex function for the *1-D* and *tree* topology due to the presence of *max* in the formulation for T_{comm}, see (Eq.3.7), and Prophet is more prone to fall into local minima.

The best results are obtained for environment M3. The reason is that the clusters in

M3 have equal communication capacity and the fastest processors will also have the fastest communication. Cluster ordering does not require the tradeoff between clusters that may offer better computational performance with clusters that may offer better communication performance. Consequently, the method is less likely to fall into local minima as discussed in Chapter 4.

In the second set of results we consider problems that have computation overlapped with communication. We expect the quality of the results to fall off slightly due to the presence of *max* in the formulation for T_c, see (Eq.4.10). The results are presented in Table 6.4-Table 6.6.

topology	**with cluster ordering % of optimal**		**random cluster ordering % of optimal**	
	5%	10%	5%	10%
ring	90.1	95.3	61.1	68.1
1-D	83.5	88.2	53.7	59.5
tree	83.9	87.0	60.0	64.6

a) No router/conversion

topology	**with cluster ordering % of optimal**		**random cluster ordering % of optimal**	
	5%	10%	5%	10%
ring	90.9	94.2	62.9	68.7
1-D	83.6	88.7	63.8	70.5
tree	85.2	89.1	66.0	70.4

b) router/conversion

Table 6.4: Simulation results for M1, hosts with unequal communication capacity. Table a) contains the simulation of metasystems with router and conversion costs included. Table b) does not simulate these costs. (overlapped communication and computation).

The performance falls off slightly for overlapped problems — about 85-90% of the runs are within 10% of optimal. To bring the performance up to the level obtained for problems without computation and communication overlap, we conjecture that a deeper exploration of the processor configuration space is needed. Some results presented in Chapter 7 indicate that the Prophet runtime overhead is sufficiently small to make a more thorough search feasible. This is the subject of future work. Over all environments and problem types the average performance is within 10% of optimal 90% of the time. We consider this acceptable performance.

Also we simulated the most common environment for M1 and M2, a single work-

topology	with cluster ordering % of optimal		random cluster ordering % of optimal	
	5%	10%	5%	10%
ring	89.8	94.3	65.4	69.2
1-D	85.7	89.7	61.9	64.5
tree	79.4	85.9	44.9	49.6

a) No router/conversion

topology	with cluster ordering % of optimal		random cluster ordering % of optimal	
	5%	10%	5%	10%
ring	91.7	95.5	51.0	55.6
1-D	82.5	86.7	53.4	57.5
tree	80.1	85.8	61.5	66.2

b) router/conversion

Table 6.5: Simulation results for M2, workstations and multicomputers. Table a) contains the simulation of metasystems with router and conversion costs included. Table b) does not simulate these costs. (overlapped communication and computation).

topology	with cluster ordering % of optimal		random cluster ordering % of optimal	
	5%	10%	5%	10%
ring	94.3	97.7	67.1	75.7
1-D	86.1	91.3	69.6	76.2
tree	87.8	90.7	74.6	78.8

a) No router/conversion

topology	with cluster ordering % of optimal		random cluster ordering % of optimal	
	5%	10%	5%	10%
ring	90.1	94.3	63.8	69.7
1-D	84.1	90.7	65.4	72.3
tree	87.6	90.1	73.8	78.0

b) router/conversion

Table 6.6: Simulation results for M3, hosts with equal communication capacity. Table a) contains the simulation of metasystems with router and conversion costs included. Table b) does not simulate these costs. (overlapped communication and computation).

station cluster and multicomputer respectively. These environments are homogeneous. In the simulation of a single processor cluster, there is no router or conversion overhead and no need for cluster ordering. The results in Table 6.7 indicate that the method handles this common case exceptionally well for all problems. Optimal elapsed times are always obtained. This agrees with the experimental results.

We have showed that on average the method performs quite well and local minima are avoided. In particular for the common cases of a single processor cluster or equal com-

topology	non-overlap % of optimal		overlap % of optimal	
	5%	10%	5%	10%
ring	100.0	100.0	100.0	100.0
1-D	100.0	100.0	100.0	100.0
tree	100.0	100.0	100.0	100.0

a) M1 - single cluster

topology	non-overlap % of optimal		overlap % of optimal	
	5%	10%	5%	10%
ring	100.0	100.0	100.0	100.0
1-D	100.0	100.0	100.0	100.0
tree	100.0	100.0	100.0	100.0

b) M2 - single multicomputer

Table 6.7: Simulation results for homogeneous environment

munication capacity, the results are extremely good for all problems. The results are also quite good for all environments if computation and communication are not overlapped. If computation and communication are overlapped, performance falls off slightly. We have also observed that the worst-case deviation from optimal is around 40% for a particular problem and this occurs very rarely. The likelihood of local minima is substantially reduced by cluster ordering and phase 2 of the partitioning method. The experimental results further substantiate that the method yields excellent performance in a practical setting.

6.3 Wide-area Parallel Processing Study

The simulation results presented here and the experimental results presented in the next chapter confirm that local-area parallel processing provides a performance benefit in many instances. The next question is whether wide-area parallel processing can be expected to deliver acceptable performance. To help answer this question we performed a simulation study of 6 network environments:

- DW (department-wide): ethernet, single router
- CW (campus-wide): ethernet, some fiber, multiple routers
- MW (metropolitan-wide)[2]: multiple routers, gateways
- NW (nation-wide)[3]: multiple gateways
- HBDW (high-bandwidth department-wide): ATM, GIGAswitch
- HBMW (high-bandwidth metropolitan-wide): Casa

2. Similar to NW, but better communication performance (measured as UVa to Sandia)
3. Measured as UVa to JPL

The first four organizations reflect today's widespread network technology and the latter two configurations reflect the new network technology that is beginning to come on-line. We model all of these environments by adjusting the routing latency and bandwidth penalty appropriately, see Table 6.8.

Network Configuration	**Routing Cost**	
	Latency (msec)	Bandwidth (msec/byte)
DW	0-1	.001
CW	1-10	.01
MW	10-100	.1
NW	100-1000	1
HBDW	0-1	.0001
HBMW	10-100	.0001

Table 6.8: Network environments

We have observed experimentally that there is approximately an order of magnitude degradation in communication capacity from DW $\rightarrow$ CW $\rightarrow$ MW $\rightarrow$ NW. The newer technologies HBDW and HBMW improve the bandwidth capacity but typically do not reduce the latency.

In our simulation study, we have simulated workstation clusters and assume a single cluster per "site". The per site cluster size ranges from 1 to 100 processors and the number of clusters (or sites) is 10. We assume that the routing penalty between *all* sites reflects the ranges in Table 6.8. A more realistic environment would have non-uniform costs between different sites, but this study is aimed at understanding the impact of network distribution so this simplifying assumption does not invalidate the results. An implementation of scheduling heuristics for wide-area environments will need to handle non-uniformity of routing costs.

We are simulating the scheduling of a single data parallel computation across multiple sites as opposed to choosing the best site. This distinction was discussed in Section 3.1.3. An important issue that we do not address here is how state information can be col-

lected in a timely manner as part of resource availability. Assuming such information could be collected in a timely manner, we are interested in whether wide-area parallel processing is feasible. We consider wide-area parallel processing to be feasible if there is a performance advantage to using remote sites.

We use the partitioning heuristic for these experiments since we have already shown it achieves performance within 10% of optimal on average and it is much faster than optimal. We simulate problems with different communication topologies containing a single computation and communication phase that may or may not have computation and communication overlapped. The problems also reflect a wide range of granularities.

To see if wide-area processing provides an advantage, we compute the average number of clusters (sites) that were selected to solve the problem. If this value is close to 1, then a single local site is adequate and wide-area parallel processing would not appear to be profitable. However it still may be the case that a remote site may be better for the problem than a local site. On the other hand, if this value is greater than 1, then wide-area parallel processing may be profitable. Clearly this value depends on problem characteristics, the most important of which is granularity. We have defined 5 granularity ranges as shown in Table 6.9. These ranges reflect the amount of computation per *PDU* per cycle.

Range	**Number of Instructions x 1000**
A	1-10
B	10-100
C	100-1000
D	1000-10000
E	10000-100000

Table 6.9: Granularity ranges

In Figure 6.4 we present the average results over all simulated problem types (*1-D*, *ring*, and *tree* topologies, overlapped and non-overlapped) for the range of granularities with respect to the network environment. Each point on the graph is the average of 45,000 experimental runs as before. The results indicate that as the distribution becomes wider-area, the impact of network distribution on site selection becomes more pronounced. How-

ever, it is also clear that as the problem granularity increases, wide-area parallel processing becomes more attractive.

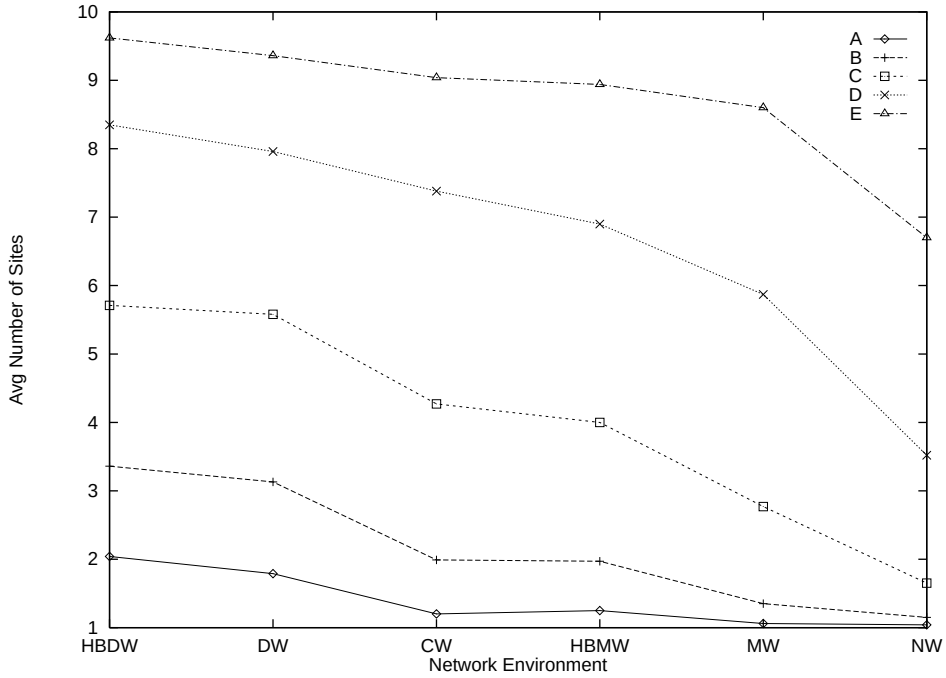


Figure 6.4: Sites vs granularity

These results indicate that there ia a point where wide-area parallel processing may become feasible. We performed a study to determine the minimum granularity requirements for problems in these environments. These experiments were designed to provide some insight into what problem sizes might be suitable for the different environments. We are interested in order-of-magnitude values for the minimum granularity. We define three granularity values of interest (m = 10 clusters):

- MIN_SITES (≈ 2 clusters)
- MID_SITES (≈ m/2 clusters)
- MAX_SITES (≈ m clusters)

MIN_SITES is the granularity at which it becomes profitable to use remote sites, MID_SITES is the granularity at which we are utilizing 50% of the sites, and MAX_SITES is the granularity at which we are utilizing 100% of the sites. The latter two values depend on the chosen value of m and are included to show that their is a point where wide-area par-

allel processing may become very attractive. For these experiments we use the same problem profile as above. In Table 6.10 we show the results obtained for MIN_SITES, MID_SITES, and MAX_SITES in the different network environments (F is the next order of magnitude beyond E). In some cases we show a range such as A-B which means that the granularity lies between the A and B ranges.

The results show that there is a point where wide-area parallel processing can be profitable. We also see an order of magnitude difference for granularity requirements for DW, CW, MW, and NW on average for MIN_SITES and MID_SITES. For MAX_SITES there is not much distinction — very large problems are required in all environments.

Granularity value	**Granularity Range**					
	HBDW	**DW**	**CW**	**HBMW**	**MW**	**NW**
MIN_SITES	A	A-B	B	B	B-C	C-D
MID_SITES	B-C	C	C-D	C-D	D	D-E
MAX_SITES	E	E	E	E-F	F	F

Table 6.10: Granularity requirements

In this chapter we have shown that the partitioning method has excellent average-case performance over a wide range of problem types and metasystem environments. Performance within 10% of optimal can be expected in the vast majority of cases. In the common environment of a single homogeneous cluster, the method always achieved optimal finishing time. These results are confirmed by the experimental results presented in the next chapter. A feasibility study into wide-area parallel processing also indicated that problems of sufficient granularity may benefit by wide-area distribution.

Chapter 7 Experimental Results

This chapter presents the experimental results that have been obtained for a suite of data parallel computations. These problems have been run in an experimental heterogeneous workstation-based environment. We show that partitioning and placement may provide a significant performance benefit, while Prophet overhead and the costs of heterogeneity, routing and data conversion, are tolerable. The benefits achieved by the use of heterogeneous processors can be large if partitioning and placement are done carefully. We describe the environment, the codes in the test suite and the experimental results obtained for each code.

7.1 Experimental Heterogeneous Environment

The experimental heterogeneous environment is a local-area ethernet-connected network of workstations. The environment contains three processor clusters: C_1 contains 6 SGI Indigo's (based on the MIPS R4000), C_2 contains 8 Sun Sparcstation 2's, and C_3 contains 8 Sun4 IPC's all joined by a router as shown in Figure 7.1. Communication between all machines is provided by MMPS [38], a reliable message-passing system based on UDP. Fortunately, there are no endian or data format differences for standard data types between these machines and no explicit conversion is needed. We have implemented a set of synthetic endian conversion routines to explore this overhead.

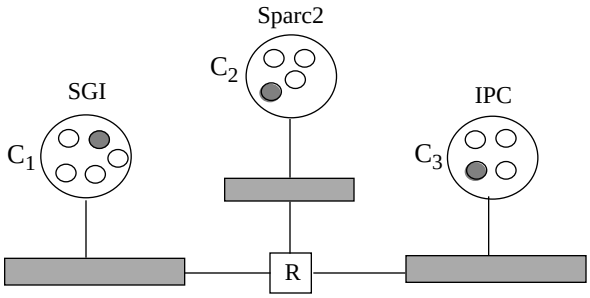


Figure 7.1: Experimental heterogeneous environment

Processor clusters in this environment exhibit heterogeneity in both computation and communication capacity. The SGI's are significantly faster than the Sparc2's and the IPC's, and the Sparc2's are faster than the IPC's in both integer and floating-point rates. The processor specifications and memory configuration are given in Table 7.1. The MMPS communication performance for the SGI's and Sparc2's are similar and both are faster than the IPC's.

Processor	**Clock Speed (Mhz)**	**Peak Mflops**	**Peak Mips**	**Memory (Mbytes)**
SGI	100	16	88	32
Sparc2	40	4	29	32
IPC	25	2	17	48

Table 7.1: Processor characteristics

All experimentation in this environment was done when the network and processors were lightly loaded. Thus, the cost functions for communication, routing, and conversion, in addition to the *arch_cost* specification presented in the next section, have been constructed under this assumption. The impact of processor load on both computation and communication costs is the subject of future work. In the presence of modest processor load, which we have defined to be a run queue length of around .33, the small inaccuracy in cost estimation did not adversely impact the quality of the results. Once a processor reached a load above this value it become a bottleneck and the accuracy of the method fell outside of acceptable bounds (10%). This is not surprising since the completion time for a

SPMD computation will be limited by the slowest worker. We discuss some strategies for dealing with load in the final chapter.

7.2 Execution Results

We present execution results for the suite of data parallel applications in the workstation-based heterogeneous environment. The results show that the scheduling framework can be successfully applied to real data parallel computations. A number of practical results are established: (1) partitioning and placement may be automated, (2) overhead is tolerable, (3) results are accurate and predictable and (4) using heterogeneous processors may provide a significant performance benefit. We show that heterogeneity is *exploited* in processor selection and data domain decomposition to gain performance, and that the primary cost of heterogeneity, conversion, can be tolerated. We also show that a secondary cost due to the distributed nature of heterogeneous resources, routing, can also be tolerated in local-area environments.

Each application is run on a range of problem instances spanning the spectrum from small- to large-grain. The results that we report are the average obtained by running each problem instance 5 times when the network and processors were lightly loaded. Some variance was observed when processor load increased or router spikes occurred. This was expected due to the amount of non-determinism inherent in network-based computing.

The metric for solution quality is elapsed time. The elapsed time is the wall-clock execution time after the workers are created and the data is distributed. The accuracy of Prophet is established by comparing the predicted T_c to the actual measured T_c. We show that the predicted T_c agrees with the actual T_c within 10%. This result validates the simulation study which is based on an accurate estimate of T_c. The simulation study indicated that the partitioning method produced results close to optimal for a given set of cost information. This cost information was used to estimate T_c.

We also establish that the Prophet runtime overhead is tolerable. The overhead presented is for a Sparc2 processor. The main program makes the Prophet calls and is run on a Sparc2. The overhead would be less on an SGI and more on an IPC. The point is to show the magnitude of the overhead term. Few optimizations have been performed within

Prophet, and these overhead values should be viewed as upper bounds. One type of optimization is the parallelization of the partitioning method in both cluster ordering and in searching the processor configuration space.

A number of comparisons are made to assess solution quality. We compare the performance obtained by Prophet to the best performance that could be obtained if a single cluster of homogeneous processors is used. This is determined by running the code for all possible number of processors within the individual processor clusters until the best was found. For small problems that require only processors in a single processor cluster, Prophet always finds the best number of processors. This result was confirmed in simulation. Small problems also highlight the importance of cluster ordering. Significant performance benefits are realized by choosing the best processor cluster. For larger problems we show that there is a benefit to using heterogeneous processors in multiple processor clusters even in the presence of conversion and routing overhead. We show that performance is superior to the best single cluster performance. However to exploit heterogeneous processors in multiple clusters, partitioning and placement must be done carefully. We also provide the best sequential time on an SGI (the fastest processor type) which is different from the time taken by the parallel code on a single processor.

Another comparison is made to determine the benefit of computing a heterogeneous data domain decomposition. For problem instances that use heterogeneous processors, we compare with the performance that is obtained when this problem is run over the same set of processors but with an equal decomposition of the data domain across all workers. We show that an equal data domain decomposition results in a load imbalance that may be substantial.

Although there is no need for conversion in this heterogeneous environment, conversion is an overhead that will impact communication cost in general. The most common form of conversion is a byte-swap endian conversion. Another comparison is made to determine the impact of byte-swap endian conversion. We have implemented an endian conversion function for each communicated data type that is used in the suite of data parallel codes. The conversion functions are enabled by setting compile-time flags. When conversion is enabled, each message is passed to the appropriate conversion routine for processing, either on the sending or receiving side. We compare performance with and

without conversion and the results indicate that even in the presence of conversion, the selected heterogeneous processor configuration out-performs the best single processor cluster, and that the conversion overhead is tolerable.

Finally a comparison is made to show benefit of automated placement. The vast majority of runtime scheduling systems (see Section 2.1.2) do not use any topology information to help guide task placement. The co-scheduling approach that Prophet has implemented can lead to much improved placement decisions. Co-scheduling collectively assigns a set of communicating workers to processors. To show the benefit of co-scheduling, we compare the performance obtained when co-scheduling is enabled to when it is not. When it is not, we use a random placement strategy that guarantees a single worker per processor. A number of different random seeds are used for each experiment and average results presented[1]. The random strategy is one of the schemes employed by the underlying Mentat scheduler [37], and is based on the load sharing model of Eager and Lazowska [21].

Co-scheduling reduces communication overhead in two ways that have been discussed in Section 4.2. However, for network-based clusters as opposed to multicomputer-based clusters, only one benefit is possible, inter-cluster placement may reduce the number of messages that cross the router. We show that for communication topologies with locality (e.g., *1-D*, and *tree*), there is a performance benefit that can be attributed to a reduced number of messages that cross the router under co-scheduling. Consequently, only problem instances that use heterogeneous processors and communicate across the router will benefit by co-scheduling in the workstation network environment.

7.3 Data Parallel Applications

We have implemented a suite of data parallel computations that test the applicability of Prophet to real codes: Gaussian elimination with partial pivoting, a canonical five-point stencil code, a large-scale finite-element code, and a gene-sequence comparison code. The latter two applications are significant codes that solve real problems in computational physics and biology respectively.

1. A single random run could simply luck into the best placement, but in general it will not.

All of these codes are structured in a style that is compatible with the SPMD computation model discussed in Chapter 3. Each code has a main program that initiates the data parallel computation, and a worker program that is appropriately parameterized to operate on a portion of the data domain. In the Mentat-Legion implementation the main program and worker program are implemented as Mentat objects. The more complex codes use additional Mentat objects that are discussed briefly in the subsequent sections.

The main program implementation is structured in the following way for all codes: (1) the domain object is created, (2) a call to Prophet is made from the main program to determine partitioning and placement, (3) the workers are created and placed on the selected processors, (4) the data domain is decomposed and passed to the workers, and (5) the computation is initiated. It is assumed that Sparc (both the Sparc2's and IPC's are Sparc processors) and SGI binaries have been compiled for each worker type[2] so that Prophet can select any processor in the heterogeneous environment for worker placement.

The performance results we present are based on elapsed time. This provides an unfiltered measure of the quality of the results and is sufficient to show the performance benefit that can be obtained with automated scheduling. Other metrics such as heterogeneous speedup [20] have also been proposed.

7.3.1 Gaussian Elimination with Partial Pivoting

Gaussian elimination with partial pivoting (GE) is perhaps the most well-known direct method for solving a linear system of equations of the form, $Ax = b$, where A is a NxN coefficient matrix, b is a right-hand-side vector, and x is the solution vector. GE is a floating-point numeric computation that contains two computations, forward reduction and backsubstitution. The forward reduction phase reduces the matrix to an upper-triangular form and is dominant with $O(N^3)$ complexity, while backsubstitution solves the upper-triangular system and has $O(N^2)$ complexity. The details of the GE algorithm may be found in [28].

2. Mentat-Legion will automatically compile binaries if they do not exist.

In the parallel implementation of GE the *PDU* is defined to be a row, and the implementation performs a row-cyclic decomposition of the matrix, see Figure 7.2. Since the amount of computation performed on a row decreases for rows further down in the matrix, a cyclic interleaving of rows gives better load balance.

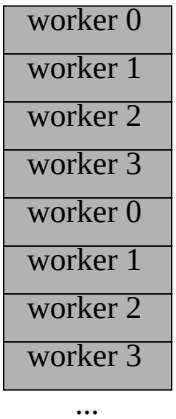


Figure 7.2: Cyclic decomposition of matrix across 4 workers

The implementation of GE arranges the workers in a broadcast topology —by convention worker 0 is the master. During partial pivoting all workers broadcast their candidate rows to the master, the master then determines the pivot row and broadcasts the selected pivot back to the workers, see Figure 7.3. The absence of locality in the broadcast topology means that placement is straightforward — the master is assigned to the cluster with the largest number of workers to minimize router traffic. Once a worker receives the current pivot it reduces the rows that it has been assigned. Forward reduction contains two phases, a broadcast communication phase for partial pivoting, and a computation phase where workers reduce their portion of the matrix. These phases are executed iteratively N-1 times. Backsubstitution is performed sequentially once forward reduction has completed. The times we report for GE are for the forward reduction phase to show the accuracy of the cost prediction for this computation.

GE has the property that the amount of communication and computation change from cycle to cycle. In this case we must provide average values for the *comp_complexity* and *comm_complexity* callbacks as discussed in Section 3.2.1. The callbacks for the dominant phases of GE (forward reduction) are given in Figure 7.4. On average a pivot row of length $N/2$ is communicated and the average instruction count is obtained by taking the

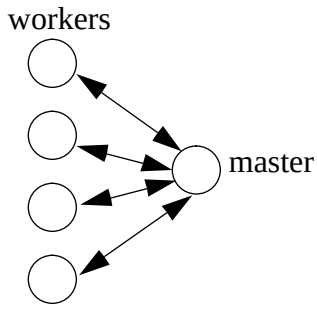


Figure 7.3: Broadcast topology for partial pivoting

total instruction count [28] and dividing by the number of *PDU*s (*N*) and cycles (*N*-1). Recall that the *arch_cost* is the problem-specific and architecture-dependent cost of computing on a *PDU* (or row) in msec/instruction[3]. We show only the *PDU* dependent component of the cost. These costs were determined by benchmarking GE on all of the machine types. On the SGI GE has the property that different problems sizes resulted in different values for *arch_cost* due to cache and memory effects. Since the callbacks can be arbitrary functions of problem parameters it is very easy to specify this type of dependence. The three values presented in Figure 7.4 refer to these cases — $N < 512$, $512 < N < 1024$, and $N > 1024$ respectively.

topology ⇒ broadcast
comm_complexity ⇒ 4(N/2) (bytes)
numPDUs ⇒ N
comp_complexity ⇒ $(2/3N^3+N(N-1))/ N(N-1)$
⇒ $(2N^2)/(3N-3)$ (fp ops)
arch_cost ⇒ SGI: = .00013, .0001, .00015
⇒ Sparc2: = .000319
⇒ IPC: = .0006

Figure 7.4: Callbacks for Gaussian elimination

GE is a basic kernel computation that poses a number of challenges. First, the amount of computation and communication vary from iteration to iteration, and the callbacks must reflect the average computation and communication per cycle. Despite the apparent inaccuracy of these callbacks, they lead to accurate cost prediction. Second, GE

3. In Chapter 3 it was described as usec/instruction but the implementation uses msec.

is a very tightly-coupled parallel computation that has a large amount of global communication. In fact, the dominant communication topology is a broadcast and the amount of communication scales linearly with the problem size and the number of processors. A global communication topology limits scalability on the network due to the limited communication bandwidth.

We ran GE on a range of matrix sizes: *N*= 256, 512, 768, 1024, and 2048, from small- to large-grained, see Table 7.2. The configuration is the number of processors in each cluster that were chosen and the PDUs are the number of *PDU*s assigned to each processor (or worker) in a particular cluster. Etime is the elapsed time taken by the problem instance. The clusters are ordered C_1 (SGI), C_2 (Sparc2), and C_3 (IPC) — this cluster ordering was determined by Prophet to be the best for all problem instances in the test suite.

Notice that as the problem size increases more processors are used as expected, but there is a hard limit. Only processors in C_1 were effectively used due to the poor scaling properties of GE on the network. There was no benefit to considering additional processors (i.e., slower Sparc2's) due to the increase in communication overhead relative to the benefit of additional processors. It is likely though that the largest problem (*N*=2048) would have benefited from additional SGI's had they been available.

Problem Size	**Configuration**			**PDUs**			**Etime (msec)**	**T_c (msec/cycle)**		**overhead (msec)**
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
256	1	0	0	256	0	0	1504	5.7	5.9	6.4
512	2	0	0	256	0	0	8891	16.2	17.4	6.8
768	3	0	0	256	0	0	21783	26.3	28.4	6.9
1024	4	0	0	256	0	0	40817	37.9	39.9	6.9
2048	6	0	0	*341	0	0	259150	118.4	126.6	7.3

Table 7.2: Experimental results for GE. The PDUs refer to the number of rows of the matrix. The entry marked * is rounded. The method gives two processors 342 PDUs, and the remaining four receive 341 (total is 2048).

The results also indicate that the method was accurate — the predicted T_c agreed

with the measured T_c often within 5% and always within 10%. This gives evidence that the use of callbacks that reflect average values can be effective. This is important because it means that the approach is not necessarily limited to problems that are extremely regular in structure. Also observe that the Prophet overhead is tolerable for GE and easily amortized as the problem size increases. At N=256 Prophet adds .4% overhead, and the overhead percentage drops off rapidly for large problems. At N=2048 Prophet adds .002% overhead.

We also present the best sequential times for GE in Table 7.3. Since the SGI is the fastest processor, we present the times for an SGI. At N=2048 the performance falls off due to memory and caching effects. The best sequential times are different from the performance obtained when the parallel code is run on one processor. The sequential code will outperform the single processor parallel code. Since Prophet is concerned with scheduling the parallel code we compare Prophet execution times for the parallel code only.

Problem Size	**Etime (msec)**
256	1743
512	13900
768	50524
1024	123053
2048	1089355

Table 7.3: Best sequential times for GE on an SGI

We have shown that the method is accurate and has small overhead, and we now show that the solution quality is quite good. Although Prophet was unable to exploit heterogeneous processors for GE, the importance of processor selection in choosing processors from C_1 first, and then in choosing the correct number of processors is demonstrated in Table 7.4. P_1, P_2 and P_3 are the best number of SGI's, Sparc2's, and IPC's respectively located by trying all possible numbers of these processors. The reported elapsed time for the best number of processors within C_1 (the SGI cluster) agrees with the predicted configuration determined by Prophet in Table 7.2. Notice that more IPC's and Sparc2's are used relative to the SGI's since they are slower and hence more balanced with respect to com-

munication. But the elapsed times indicate that the use of fewer faster SGI's leads to superior performance. There is no predictable pattern as to how much the performance will increase, it depends on the problem size, how many processors were used, and the computation and communication capabilities of the processors. What can be said is that the performance increase is substantial.

Problem Size	**Best P_1 and Elapsed Time (msec)**		**Best P_2 and Elapsed Time (msec)**		**Best P_3 and Elapsed Time (msec)**		**% Benefit of Prophet configuration with respect to best single cluster performance**		
	P_1	Etime	P_2	Etime	P_3	Etime	C_1	C_2	C_3
256	1	1504	1	3774	2	6350	---	151%	322%
512	2	8891	1	11957	5	27134	---	35%	205%
768	3	21783	6	37506	6	64735	---	84%	197%
1024	4	40817	7	70485	8	133604	---	66%	227%
2048	6	259150	8	525610	8	858102	---	102%	231%

Table 7.4: Best performance for GE

GE is also able to tolerate endian conversion fairly easily, see Table 7.5. All workers convert their candidate pivot before sending to the master worker during partial pivoting. This allows the workers to perform conversions in parallel. Conversion increases the per cycle elapsed time by a few percent. At *N*=512, we observe a larger increase of 7% that we

Problem Size	**Configuration**			**PDUs**			**Etime (msec)**	**T_c (msec/cycle)**		**% increase in T_c**
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
256	1	0	0	256	0	0	1504	5.7	5.9	---
512	2	0	0	256	0	0	9556	16.5	18.7	7%
768	3	0	0	256	0	0	21936	26.7	28.6	1%
1024	4	0	0	256	0	0	41432	38.5	40.5	2%
2048	6	0	0	*341	0	0	265086	119.6	129.5	2%

Table 7.5: Impact of endian conversion for GE

speculate is due to cache effects. The addition of conversion does not significantly change the overhead experienced by Prophet. Also observe that the estimation of T_c in Table 7.5

reflects the added conversion cost and is still very accurate.

7.3.2 Five-Point Stencil

The canonical stencil computation is a common data parallel problem that appears in a number of different application areas including image processing and iterative PDE solvers. The stencil computation is based on a underlying grid that arises from a spatial decomposition of the problem. This decomposition is often a discrete representation of a continuous domain. The values computed at the grid points and the relationship among grid points are different for different problem domains. In a stencil computation the values computed at a grid point are dependent on the values computed at neighboring grid points. In image processing problems the grid points refer to pixels of the image while for PDEs the grid points refer to points in the spatial domain of the problem. For example, in the PDE that arises from modeling heat flow along a metal plate, the grid points would correspond to points on the surface of the plate.

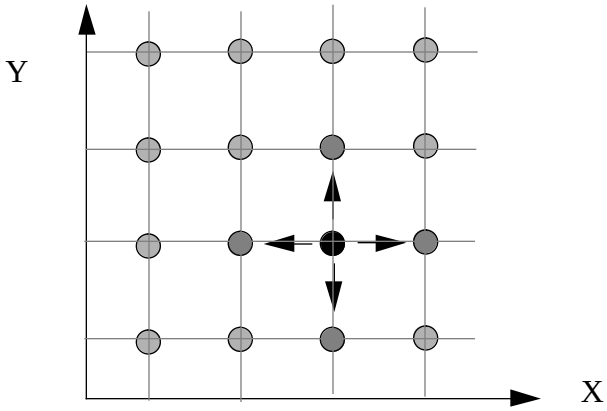


Figure 7.5: 2-D grid

Perhaps the simplest stencil computation is the five-point stencil that arises from the discretization of PDEs in two variables, see Figure 7.5. Each point is coupled with it's north, south, east, and west neighbors as shown for the black point. Points on the boundary require some type of boundary conditions to help resolve their value. During the stencil computation values associated with these points are repeatedly updated until some

convergence or stopping criteria is met. The size of the grid reflects the level of fidelity and accuracy that is desired. A larger grid has a finer resolution and is more accurate, but requires additional computation and memory.

We have implemented a five-point stencil code (STEN) for an iterative PDE solver that can be used to solve Laplace's equation: $u_{xx} + u_{yy} = 0$ on the unit square. Using finite-differences a grid is imposed over this domain with the grid points u_{ij} related in the following way: $-u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} + 4u_{i,j} = 0, i, j = 1, \ldots, N$. A grid of size N produces a linear system that contains N^2 equations corresponding to N^2 interior grid points. We solve this system using Jacobi's method [28]. This algorithm has a large amount of inherent parallelism and has much better scaling properties than does GE. Both GE and STEN have a computation granularity that scales well with problem size N — N^3 and N^2 respectively. But the dominant communication pattern in STEN is a local nearest-neighbor exchange of grid point values that has better scaling properties than the global communication required in GE.

In the parallel implementation of STEN the *PDU* is defined to be a row of an *N*x*N* grid, and the workers are arranged in a *1-D* communication topology as shown in Figure 3.8. Unlike GE, STEN has locality and placement assigns workers to processors in order to preserve locality for the *1-D* topology. Each worker receives a row-contiguous share of the grid that is proportional to the power of the processor to which it has been assigned. The same amount of computation is performed on each row of the grid (except the boundaries) so a cyclic decomposition is unnecessary. The workers execute a single dominant computation phase where the grid point values are updated according to the rule given above, followed by a dominant communication phase where the workers exchange north and south borders of the grid. These phases are executed iteratively until some stopping criteria. We run STEN for 100 iterations and report the elapsed time.

STEN is a regular floating-point computation and the callbacks for STEN are given in Figure 7.6. Notice that the callbacks are simpler than GE and this reflects the regular

topology ⇒ 1-D
comm_complexity ⇒ 4N (bytes)
numPDUs ⇒ N
comp_complexity ⇒ 5N (fp ops)
arch_cost ⇒ SGI: = .0001, .00015
⇒ Sparc2: = .000319, .00028
⇒ IPC: = .0006, .00072

Figure 7.6: Callbacks for stencil

nature of this computation. The same amount of computation and communication are performed in each iteration and the amount of computation per *PDU* (or row) is the same across the entire data domain. Again we show only the *PDU* dependent portion of *arch_cost*. STEN has the property that different problems sizes resulted in different values for *arch_cost* due to cache and memory effects. The values presented above refer to these cases — $N < 1024$ and $N > 1024$ respectively. Unlike GE, the Sparc2 and IPC also exhibited a sensitivity to problem size.

Unlike GE, STEN has much better scaling properties and is able to exploit heterogeneous processors. The dominant communication topology of STEN is a *1-D* which is a class of nearest-neighbor topologies that tend to scale well.

We ran STEN on a range of grid sizes: N= 64, 128, 256, 512, 1024, and 2048, from small- to large-grained for 100 iterations. The number of iterations selected does not affect the per cycle elapsed times, but the larger the number of iterations the more easily Prophet overhead may be amortized over the entire computation. The first set of results are given in Table 7.6 and are qualitatively similar to the results for GE in Table 7.2.

Observe that the predicted T_c is still within 10% of the actual T_c. Also note that for larger problems the method computes a heterogeneous data domain decomposition with a different number of *PDU*s assigned to workers on different processor types. The overhead is a little higher than for GE since Prophet explores more processors and clusters. But the overhead is still easily amortized. At N=64 Prophet adds 4% overhead, and the overhead percentage drops off rapidly for large problems. At N=2048 Prophet adds .03% overhead.

Problem Size	Configuration			PDUs			Etime (msec)	T_c (msec/cycle)		overhead (msec)
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
64	1	0	0	64	0	0	200	2.1	2.0	7.3
128	3	0	0	*43	0	0	776	7.3	7.8	6.8
256	4	0	0	64	0	0	1620	16.1	16.3	7.2
512	6	8	0	*61	18	0	4390	38.4	43.9	10.5
1024	6	8	5	*110	34	18	9635	95.1	96.4	10.2
2048	6	8	6	*178	95	36	36558	346.8	365.6	10.7

Table 7.6: Experimental results for STEN. The PDUs refer to the number of rows of the grid. The entry marked * is rounded as appropriate, e.g. for N=128 the method gives the processors 43, 43, and 42 PDUs respectively.

Prophet begins to use heterogeneous processors at *N*=512 when the computation granularity becomes large enough to offset the communication overhead. At *N*=1024 the problem is big enough to warrant the use of processors in all clusters.

We present the best sequential times for STEN on an SGI in Table 7.3 (shown also for 100 iterations). Note that the best sequential time for *N*=128 is better than the best time the parallel code can achieve. This is not surprising since the sequential code uses statically allocated arrays while the parallel code uses dynamic data structures.

Problem Size	Etime (msec)
64	174
128	698
256	2924
512	13287
1024	51550
2048	282984

Table 7.7: Best sequential times for STEN on an SGI

To assess the performance of the selected configuration, we present the best elapsed times observed when only a single cluster is used, see Table 7.8. The results show two things. First, as with GE, Prophet chooses the best number of processors to use when a single processor cluster is selected. Second, the use of heterogeneous processors provides a performance benefit over the use of a single processor cluster for *N*=512, 1024, and 2048.

Problem Size	**Best P_1 and Elapsed Time (msec)**		**Best P_2 and Elapsed Time (msec)**		**Best P_3 and Elapsed Time (msec)**		**% Benefit of Prophet configuration with respect to best single cluster performance**		
	P_1	Etime	P_2	Etime	P_3	Etime	C_1	C_2	C_3
64	1	200	1	556	1	1161	---	178%	481%
128	3	776	6	1186	8	2433	---	53%	216%
256	4	1620	8	2333	8	4473	---	44%	176%
512	6	4840	8	6677	8	11377	9%	52%	159%
1024	6	12075	8	23046	8	47835	25%	139%	396%
2048	6	61295	8	84032	8	218650	38%	122%	478%

Table 7.8: Best performance for STEN

A key element in achieving good performance is a heterogeneous data domain decomposition that gives processor load balance. To show the benefit of a heterogeneous data domain decomposition, we show the results of running STEN across the heterogeneous configurations selected at *N*=512, 1024, and 2048, but with an equal decomposition of the data domain in which all processors receive an equal share of *PDU*s, see Table 7.9. The load imbalance causes a performance degradation that is significant for large problems, as much as 89% for STEN. The precise performance impact of the imbalance is difficult to predict and is problem-dependent, but load imbalance can cause a performance degradation that can be severe. In fact, the load imbalance completely eliminates the benefit of using heterogeneous processors and reduces the effective parallelism. For example, for *N*=512, 1024, and 2048, it would have been better to use 6 SGI's than to use the selected configuration with an equal data domain decomposition, see Table 7.8.

Problem Size	**Configuration**			**PDUs**			**Elapsed Time (msec)**	**% Increase in Etime with respect to balanced load**
	C_1	C_2	C_3	A_1	A_2	A_3		
512	6	8	0	*36	36	0	5125	17%
1024	6	8	5	*54	54	54	18201	89%
2048	6	8	6	*102	102	102	64903	77%

Table 7.9: Benefit of heterogeneous data domain decomposition for STEN

Although the load imbalance results show that a large performance degradation occurs, we might expect an even larger degradation. For example at N=2048 each IPC is given 36 rows when load balanced vs. 102 rows when not load balanced, so we might expect a performance degradation of over 100% due to an increase in computation time. However the load imbalance only impacts the computation part of T_c and so the increase in T_c depends on the T_{comp} and T_{comm} components of T_c. For example if T_{comm} were 0 then we would expect to see a degradation over 100%. However if computation and communication costs were more balanced then we would expect a smaller degradation which is consistent with the results we have obtained.

The use of multiple processor clusters for N=512, 1024 and 2048 also indicates that router overhead is worth paying for the gain in communication bandwidth and computation cycles. We also show that endian conversion is tolerated by STEN in a manner similar for GE, see Table 7.10. Conversions are performed when the workers receive border rows from their north and south neighbors. The rows are single precision floating-point numbers. The workers perform the endian conversions in parallel. Conversion adds very small overhead and does not alter the use of heterogeneous processors. Prophet still chooses heterogeneous processors even with a conversion penalty, and the resulting elapsed times are still superior to the best single cluster elapsed times.

Problem Size	**Configuration**			**PDUs**			**Etime (msec)**	**T_c (msec/cycle)**		**% increase in T_c**
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
64	1	0	0	64	0	0	200	2.1	2.0	---
128	3	0	0	*43	0	0	790	7.4	7.9	1%
256	4	0	0	64	0	0	1649	16.4	16.5	1%
512	6	8	0	*61	18	0	4579	39.3	45.8	4%
1024	6	8	5	*110	34	18	10386	96.2	103.8	8%
2048	6	8	6	*178	95	36	38882	349.3	388.3	6%

Table 7.10: Impact of endian conversion for STEN

Finally we show that the co-scheduling model of Prophet provides a significant per-

formance improvement in the event that multiple processor clusters are selected. We ran STEN using the same configuration and data domain decomposition computed by Prophet but with a random placement that assigns a single worker per processor see Table 7.11. Under a random assignment workers in the *1-D* topology may have north and south neighbors in other processor clusters. Thus, the amount of communication that crosses the router will increase. The router congestion contributed to a large increase in elapsed time for the problem instances. The co-scheduling results are problem-dependent and also depend on the random assignments that were used. Nonetheless, we assert that co-scheduling is superior to the alternative of not using topology information and that the performance benefit may be large.

Problem Size	**Configuration**			**PDUs**			**Elapsed Time (msec)**	**% Increase in Etime with respect to co-scheduling**
	C_1	C_2	C_3	A_1	A_2	A_3		
512	6	8	0	61	18	0	6483	48%
1024	6	8	5	110	34	18	17842	76%
2048	6	8	6	178	95	36	63628	74%

Table 7.11: Benefit of co-scheduling for STEN

7.3.3 Finite-Element Computation

Finite-element methods have been widely used for problems in structural mechanics and more recently in electromagnetic-scattering (EM) problems. Finite-elements can effectively model the specific geometry of an object by unstructured gridding, see Figure 7.7. In the EM problem an electromagnetic wave illuminates a set of objects (scatterers) and the electromagnetic field scattered from the objects is calculated. The ability of finite-elements to accurately model the scatterer's surface makes the finite-element method attractive for such problems.

We have implemented a 2D version of EM problem which solves for the electromagnetic fields in the vicinity of a set of scatterers, see Figure 7.8. The code solves a Helm-

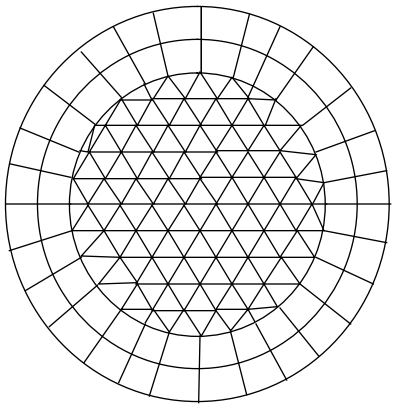


Figure 7.7: A Simple finite element mesh

holtz equation with an absorbing boundary condition defined on the boundary Γ that uniquely specifies the problem. A description of the 2D integral equation can be found in [92]. A finite-element mesh is imposed on the problem and the 2D integral equation is transformed into a system of linear equations. The problem domain is meshed with nodal points that match the geometry of the objects and the electromagnetic field values are computed at these points. In the 2D EM problem the node geometries are triangles or quadrilaterals.

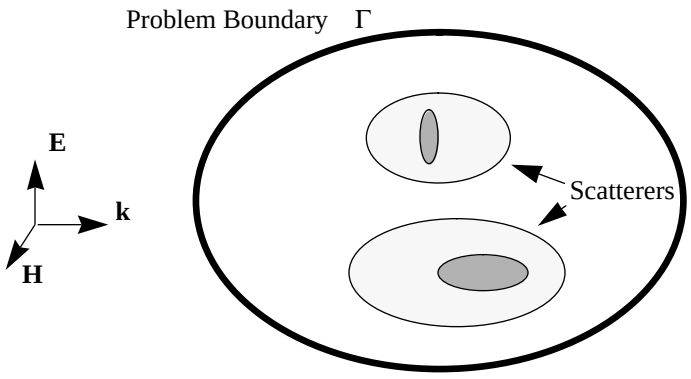


Figure 7.8: The general 2D EM scattering problem

The EM problem reduces to solving a linear system of equations of the form: $\bar{\mathbf{K}} \cdot \mathbf{d} = \mathbf{F}$, where $\mathbf{d}$ is the vector field, $\bar{\mathbf{K}}$ is the stiffness matrix, and $\mathbf{F}$ is the force vector. The computation of $\bar{\mathbf{K}}$ and $\mathbf{F}$ depend on the nodal basis functions and are discussed

in [92]. The elements of $\overline{\mathbf{K}}$ and $\mathbf{F}$ are complex numbers. The finite-element (FEM) computation is a large-scale 3500-line code that contains two coupled data parallel computations that are executed sequentially, *assembly* and *solve*. In the *assembly* phase the stiffness matrix $\overline{\mathbf{K}}$ and the force vector $\mathbf{F}$ are computed. The stiffness matrix that results is large, very sparse, and symmetric. Fortunately it has small bandwidth relative to the size of the matrix. The *solve* computation uses a bi-conjugate gradient solver BCG to solve the system. BCG is known to have instability problems but we did not encounter this behavior in our experiments. The stiffness matrix is first preconditioned by diagonal scaling to improve convergence.

In *assembly* the finite-element mesh is decomposed across a set of workers that compute contributions to the stiffness matrix and force vector. Each *assembly* worker receives a number of elements that are proportional to the power of the processor to which it has been assigned. For each contained element a stiffness matrix contribution is computed. Elements on the problem boundary contribute to the force vector as well. In *solve* the stiffness matrix and a set of vectors computed by BCG are decomposed across a set of *solve* workers. These computations are coupled — the *assembly* workers send their stiffness matrix contributions directly to the appropriate *solve* workers, see Figure 7.9. Prophet is first applied to the *solve* phase in order to determine the placement and identity of the *solve* workers. This must be done first since the *assembly* workers need to know where to transmit their stiffness matrix contributions. Once the *solve* workers are known, Prophet is applied to the *assembly* phase.

The *assembly* phase is straightforward with a single dominant computation and communication phase operating over the domain of finite-elements. Computing the stiffness matrix is dominant over the force vector. The finite-elements are randomized for load balance (some element types require more computation) and distributed to the *assembly* workers. Each *assembly* worker computes a stiffness matrix contribution for each contained element and transmits a list of such values to the appropriate *solve* worker. Because

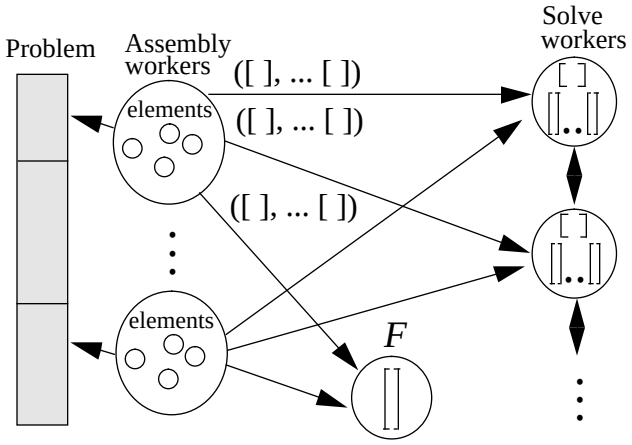


Figure 7.9: Parallel finite element computation

Prophet is first applied to the *solve* phase, the identity of the *solve* workers and matrix decomposition are known to the *assembly* workers. *Assembly* is an iterative computation with the workers computing and storing stiffness matrix values in a set of bins each corresponding to a *solve* worker. At the end of each iteration the *assembly* workers send the bin contents to the *solve* workers. The stiffness matrix is never stored in a single place, it is kept distributed across the *solve* workers. The number of iterations is dependent on the number of elements in the problem. Collectively the communication topology is a broadcast. The callbacks for the *assembly* phase are shown in Figure 7.10. The functions are more complex

topology ⇒ broadcast
comm_complexity ⇒ ((num_nodes^2)/w)*k_entry_size*(num_elmts/cycles) (bytes)
numPDUs ⇒ num_elmts
comp_complexity ⇒ 124(num_nodes^2)+30(num_nodes^2+1)*(num_elmts/cycles)
arch_cost ⇒ SGI: = .00017
⇒ Sparc2: = .000335
⇒ IPC: = .00078

Figure 7.10: Callbacks for finite-element code (*assembly*)

than for GE or STEN and depend on several problem parameters, *num_nodes*, the number of nodes per finite-element, *num_elmts*, the number of elements in the problem domain, *cycles*, the number of iterations, *w*, the number of *solve* workers, and *k_entry_size*, the size in bytes of a single stiffness matrix value. These parameters are marshaled into PV and used

by the appropriate callback functions. The problem instances that we have used contain either 3 point triangle or 9 point quadrilateral elements containing 3 and 9 nodes respectively. The callbacks for *comm_complexity* and *comp_complexity* are computed as average values over all elements and cycles much like GE.

The *solve* phase is much more complex. It highlights a limitation of the use of dominant phases to guide partitioning and placement. Although *solve* has a dominant sparse matrix-vector multiplication and dot-product, we have observed that for small problem sizes (all of our problem instances are relatively small), the other phases must be considered since the dominant computation does not dominate the sum total of the other phases. The other phases include a number of global tree communications to compute the constants alpha and beta, several global dot products, and the residual in BCG. Because the callbacks may be arbitrary functions it is easy to specify that all phases are to be considered by Prophet.

For simplicity we present the callbacks for the sparse matrix-vector multiplication and dot product ($A\vec{P}$, $A\vec{P}$) only, see Figure 7.11. The amount of computation depends on the average number of non-zeros, *nnz*, per row in the stiffness matrix. The workers are arranged in a *1-D* communication topology to exchange portions of the $\vec{P}$ vector needed for the matrix-vector multiply as shown in Figure 7.9. The FEM problem instances result in small bandwidth, *bw*, and only a small amount of communication is required between workers to establish the local $\vec{P}$ vector needed to compute $A\vec{P}$. The *N*x*N* stiffness matrix is decomposed

topology ⇒ 1-D
comm_complexity ⇒ 16*bw (bytes) //16 is the size of a complex number
numPDUs ⇒ N
comp_complexity ⇒ nnz*6 + 8 (fp ops) //nnz*6 is for $A\vec{P}$, 8 is for the dot product
arch_cost ⇒ SGI: = .0001, .00017
⇒ Sparc2: = .000335, .000435
⇒ IPC: = .00078

Figure 7.11: Callbacks for finite-element code (*solve*)

into contiguous rows across the workers and the *PDU* is a row of the matrix. Each *solve* worker receives a number of rows that are proportional to the power of the processor to which

it has been assigned. As for STEN and GE, the *arch_cost* may change for different problem sizes. We present the elapsed times for *assembly* and *solve* separately.

FEM presents the most important challenge to our approach. It contains two coupled data parallel computations, *solve* and *assembly,* that operate over two data domains. We have applied Prophet to three instances of the finite-element problem, *dct3*, containing 2160 3 point triangle elements, *dcq9*, containing 2304 9 point quadrilateral elements, and *dcq9x2*, a synthetic version of *dcq9* that results in a 2x2 matrix with sub-matrices each corresponding to the *dcq9* stiffness matrix.

Both *dct3* and *dcq9* are real instances of an electromagnetic scattering problem provided by Nasa-JPL [92]. The input files contain a discretization of the problem domain for a specific EM problem instance. The stiffness matrix sizes are *N*=1117 for *dct3*, *N*=9303 for *dcq9,* and *N*=37057 for *dcq9x2*. These problem instances are very sparse with the average number of non-zeros per row: 10 for *dct3,* 26 for *dcq9,* and 104 for *dcq9x2*. Fortunately, the matrix bandwidth is fairly small and requires little communication: 44 for *dct3,* 248 for *dcq9,* and 490 for *dcq9x2*[4]. We present the initial set of results for FEM in Table 7.12.

Problem	**Configuration**			**PDUs**			**Etime (msec)**	**T_c (msec/cycle)**		**overhead (msec)**
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
dct3-assembly	4	0	0	540	0	0	1153	49.4	53.4	10.8
dct3-solve	1	0	0	1117	0	0	2913	28.1	27.8	10.5
dc9q-assembly	6	4	0	*287	145	0	2905	134.8	126.1	10.1
dcq9-solve	4	0	0	*2328	0	0	48410	115.5	124.4	13.4
dcq9x2-assembly	6	4	0	*1148	582	0	9660	103.2	105.0	7.8
dcq9x2-solve	6	8	0	*4243	1657	0	272079	676.7	701.0	15.3

Table 7.12: Experimental results for FEM. The PDUs for *assembly* refer to the number of elements and for *solve*, the number of rows of the stiffness matrix. Problem dct3 required 105 iterations for *solve* and 22 iterations for assembly; dcq9 required 388 iterations for *solve* and 23 iterations for assembly; and dcq9x2 required 388 iterations for *solve* and 92 iterations for assembly. The entries marked * were rounded to the nearest integer.

4. The bandwidth for *dcq9x2* has been optimized by equation reordering.

The *solve* and *assembly* phases are handled separately by Prophet. These phases operate in different data domains, the *PDU*s given for *assembly* are the number of elements, and for *solve*, the number of rows of the stiffness matrix. These results follow the pattern established by GE and STEN. Small problems such as *dct3* have small computation granularity for both *assembly* and *solve* and cannot effectively use many processors. Bigger problems such as *dcq9* and *dcq9x2* are able to more effectively use additional processors due to larger computation granularity. The *solve* phase is tightly-coupled and sparse and can only use heterogeneous processors for *dcq9x2*. We also show that the method is accurate — T_c is within 10% of the measured elapsed time and that overhead is small. The overhead contributes less than 1% of the elapsed time and is easily amortized. The accuracy for *solve* was notable because it is not simply based on the dominant phases, but it is based on the sum of a number of communication and computation sub-phases. We present the best sequential times for FEM on an SGI in Table 7.13.

Problem	**Etime (msec)**
dct3-assembly	1383
dct3-solve	2531
dc9q-assembly	13422
dcq9-solve	73712
dcq9x2-assembly	53544
dcq9x2-solve	1642404

Table 7.13: Best sequential times for FEM on an SGI

The performance results were quite good when compared to the best single cluster elapsed times, see Table 7.14. When Prophet chose a single processor cluster, it selected the best number of processors for *dct3* and *dcq9-solve*. In the other cases *dcq9-assembly* and *dcq9x2*, the use of heterogeneous processors provided a performance improvement over the best single cluster times. The latter results depend on a heterogeneous data domain decomposition for load balance.

We show the results of running FEM across the heterogeneous configurations for *dcq9-assembly* and *dcq9x2* with an equal decomposition of the data domain, see Table 7.15. We

Problem	**Best P_1 and Elapsed Time (msec)**		**Best P_2 and Elapsed Time (msec)**		**Best P_3 and Elapsed Time (msec)**		**% Benefit of Prophet configuration with respect to best single cluster performance**		
	P_1	Etime	P_2	Etime	P_3	Etime	C_1	C_2	C_3
dct3-assembly	4	1153	8	1468	8	2053	--	27%	78%
dct3-solve	1	2913	1	6570	3	11527	---	125%	296%
dc9q-assembly	6	3372	8	4609	8	10259	16%	59%	253%
dcq9-solve	4	48410	8	87276	8	195955	---	80%	305%
dcq9x2-assembly	6	11304	8	14166	8	34226	17%	47%	254%
dcq9x2-solve	6	305131	8	574628	8	1017134	12%	111%	274%

Table 7.14: Best performance for FEM

obtained results similar to that for STEN, namely, the performance degradation due to load imbalance can be large and has the effect of eliminating the effective parallelism. For example, the performance of *dcq9-assembly* and *dcq9x2* was better using 6 SGI's than the selected configuration with an equal data domain decomposition, see Table 7.14.

Problem	**Configuration**			**PDUs**			**Elapsed Time (msec)**	**% Increase in Etime with respect to balanced load**
	C_1	C_2	C_3	A_1	A_2	A_3		
dc9q-assembly	6	4	0	*230	230	0	4862	67%
dcq9x2-assembly	6	4	0	*921	921	0	12916	34%
dcq9x2-solve	6	8	0	*3706	3706	0	536023	94%

Table 7.15: Benefit of heterogeneous data domain decomposition for FEM

The impact of endian conversion on FEM was also minimal. During the *assembly* phase the workers convert their data in parallel before sending to the *solve* workers. The data contains a list of stiffness matrix entries each containing two integer matrix indices and a complex matrix value, two double precision floating-point numbers. During the *solve* phase the *solve* workers convert the local $\vec{P}$ vector contributions in parallel upon receipt from their north and south neighbors. The $\vec{P}$ vector elements are complex numbers. As with STEN conversion adds very small overhead and does not alter the use of heterogeneous processors. Prophet still chooses heterogeneous processors even with a conversion penalty, and the resulting elapsed

Problem	Configuration			PDUs			Etime (msec)	T_c (msec/cycle)		overhead (msec)
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
dct3-assembly	4	0	0	540	0	0	1195	50.7	55.3	4%
dct3-solve	1	0	0	1117	0	0	---	---	---	---
dc9q-assembly	6	4	0	*287	145	0	3279	137.1	142.3	6%
dcq9-solve	4	0	0	*2328	0	0	49152	116.6	126.3	2%
dcq9x2-assembly	6	4	0	*1148	582	0	10061	105.4	109.0	4%
dcq9x2-solve	6	8	0	*4243	1657	0	275064	678.7	707.0	1%

Table 7.16: Impact of endian conversion for FEM

times are still superior to the best single cluster elapsed times, see Table 7.16.

We show the benefit of co-scheduling for *solve* in which the dominant communication topology is a *1-D* topology. The *assembly* phase uses a *broadcast* that does not exhibit locality. On the other hand, the *1-D* topology has locality and co-scheduling will reduce the number of messages that cross the router. The single problem instance for *solve* that uses heterogeneous processors with co-scheduling disabled is shown in Table 7.17. We see that co-scheduling provides a performance benefit.

Problem	Configuration			PDUs			Elapsed Time (msec)	% Increase in Etime with respect to co-scheduling
	C_1	C_2	C_3	A_1	A_2	A_3		
dc9q-assembly	6	4	0	287	145	0	---	---
dcq9x2-assembly	6	4	0	1148	582	0	---	---
dcq9x2-solve	6	8	0	4243	1657	0	361840	31%

Table 7.17: Benefit of co-scheduling for FEM

7.3.4 Biological Sequence Comparison

Biological sequence comparison is concerned with the classification of protein sequences that have been determined by DNA cloning and sequencing techniques. Because it is difficult to determine the function of a given protein, a newly sequenced protein is compared with other proteins that have evolved from a common ancestor. The idea being that if the pro-

tein in question is similar to an enzyme whose function is known then it is likely that this protein performs a similar function [30].

DNA and protein molecules are composed of four nucleotide base pairs (A, C, G, T) that form the building blocks for DNA and the 20 amino acids for proteins. Comparing protein or DNA sequences is a string matching problem over strings of base pairs. Through the Human Genome Initiative, DNA and protein libraries are available for most published sequences. The comparison problem is a computationally intensive process that is well-suited to parallel execution.

We have implemented a parallel sequence comparison code, Complib, that compares a source library of sequences to a target library of sequences. Complib, like FEM, is a real code that is 6000 lines of C++ code. Unlike the other codes, Complib is a non-floating point computation and is more loosely-coupled than GE, STEN or FEM. Like GE, Complib contains global communication but the computation granularity is large enough to enable this code to scale very well. Complib utilizes three heuristics for string matching, Smith-Waterman, a rigorous dynamic programming algorithm, Fasta, a fast heuristic that improves performance 20-100 times, and Blast, another fast heuristic. We have experimented with Smith-Waterman (SW) and Fasta (FA) on a set of input libraries that are randomized for load balance. The details of these algorithms may be found in [30].

In the parallel implementation of Complib (CL), the target library is decomposed across a set of workers. Each worker compares all of the sequences it is assigned to a sequence in the source library during a single iteration. The workers are arranged in a tree with the leaves performing the computation. Complib is an example of the *hybrid-tree* topology discussed earlier. Each leaf worker receives a number of target sequences that are proportional to the power of the processor to which it has been assigned. A set of interior nodes are responsible for fanning the source sequences down to the leaves for sequence comparison and fanning the comparison results from the leaves back up the tree to a recorder object, see Figure 7.12. The results contain a comparison score for the current source sequence generated

by each worker based on the target sequences. The amount of data in the result list is pro-

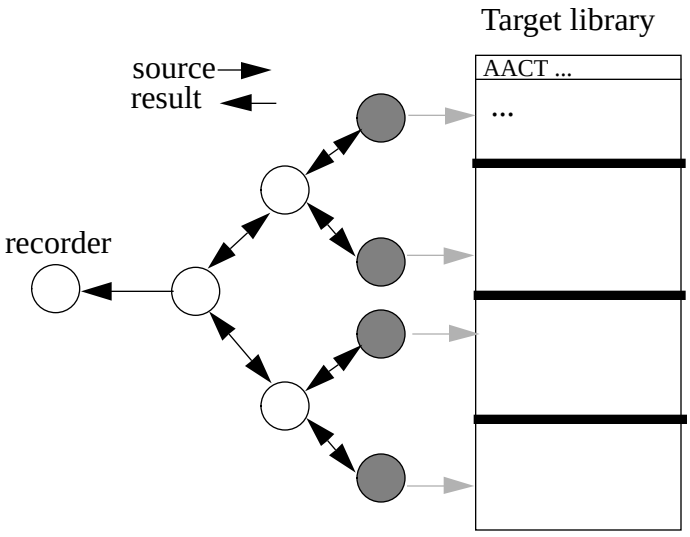


Figure 7.12: Parallel sequence comparison

portional to the number of target sequences.

The structure of this computation is straightforward. CL has a single dominant computation for sequence comparison, and a dominant communication where results are communicated up the tree. The callbacks for CL are given Figure 7.13. The *PDU* is a target sequence. The callbacks depend on two problem parameters, the number of target sequences, *num_target_sequences*, and *w*, the number of workers in the comparison tree. The comparison record is 16 bytes and the *log* term is the height of the tree. The *comm_complexity* is the average size of a result message transmitted by a worker.

Since the amount of computation per *PDU* (target sequence) does not depend on problem parameters, we specify a simpler callback for *comp_complexity.* We define *comp_complexity* to be 1 such that when it is multiplied by *arch_cost* it returns the real computation cost, see (Eq.4.1). The *arch_cost* shown is for the FA and SW comparison algorithms respectively. SW is extremely compute-intensive relative to FA. We present completion times for both SW and FA for the sequence comparison portion of the computation.

topology ⇒ hybrid_tree
comm_complexity ⇒ (16*$\log_2$ (w) * num_target_sequences)/w (bytes)
numPDUs ⇒ num_target_sequences
comp_complexity ⇒ 1 (fp ops)
arch_cost ⇒ SGI: = .9, 90.0
⇒ Sparc2: = 4.2, 220.0
⇒ IPC: = 8.4, 880.0

Figure 7.13: Callbacks for CL

CL is another real code like FEM. It has the nice property that it is loosely-coupled and it tends to scale better with processors than the other codes. We have applied Prophet to two different versions of CL, one that uses Fasta (FA) and another that uses Smith-Waterman (SW). We experimented with five problem instances, two for Fasta, *FA-1* and *FA-2*, and three for Smith-Waterman, *SW-1*, *SW-2*, and *SW-3*. A problem instance is defined by a particular target library that is decomposed across the CL workers. In all cases the same source library is used, a library containing 1439 sequences. This corresponds to the number of cycles or iterations. The target library sizes are the following: 287 sequences for *FA-1*, 4397 sequences for *FA-2*, 144 sequences for *SW-1*, 620 sequences for *SW-2*, and 1439 sequences for *SW-3*. All libraries have been randomized to help insure load balance when the target library was distributed across the workers.We present the results for CL in Table 7.18.

Problem	**Configuration**			**PDUs**			**Etime (sec)**	**T_c (msec/cycle)**		**overhead (msec)**
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
FA-1	4	0	0	72	0	0	151	100.6	105.4	12.1
FA-2	6	5	0	*622	133	0	1637	1152.1	1137.7	14.5
SW-1	6	0	0	24	0	0	3646	2438.4	2533.9	11.2
SW-2	6	8	0	82	16	0	11594	7513.8	8057.3	15.7
SW-3	6	8	8	*170	34	17	22807	15618.4	15849.5	15.6

Table 7.18: Experimental results for CL. The number of entries in the source library (iterations) for all problems was 1439. The PDUs refer to the number of target sequences. The entries marked * are rounded as appropriate.

For all problem instances Prophet is accurate and overhead is small relative to total elapsed time[5]. Also observe that Smith-Waterman has a much larger computation granularity and is able to more effectively exploit additional processors. We present the best sequential times for CL on an SGI in Table 7.19. The entries marked with a ** were estimated due to the projected length of the run. We estimated the total elapsed time based on the per cycle elapsed time observed after 100 iterations and multiplied by the number of iterations, 1439. Since the libraries are randomized this should be an accurate estimator for the entire problem.

Problem	**Etime (sec)**
FA-1	372
FA-2	5695
SW-1	18649**
SW-2	80296**
SW-3	185069**

Table 7.19: Best sequential times for CL on an SGI

The performance results for CL were also good when compared to the best single cluster elapsed times, see Table 7.20. In particular the use of heterogeneous processors provided a significant performance improvement over the best single cluster times. Again the entries marked ** were estimated based on 100 iterations.

Problem	**Best P_1 and Elapsed Time (sec)**		**Best P_2 and Elapsed Time (sec)**		**Best C_3 and Elapsed Time (sec)**		**% Benefit of Prophet configuration with respect to best single cluster performance**		
	P_1	Etime	P_2	Etime	P_3	Etime	C_1	C_2	C_3
FA-1	6	151	8	389	8	770	---	157%	409%
FA-2	6	1682	8	4324	8	7705	3%	164%	371%
SW-1	6	3646	8	12331	8	29990**	---	238%	722%
SW-2	6	16211	8	72777**	8	139410**	40%	527%	1102%
SW-3	6	30152	8	127073**	8	242851**	32%	457%	964%

Table 7.20: Best performance for CL

5. Unlike the other tables the units of time for CL are in seconds.

The large improvement result for SW is due, in part, to the loosely-coupled structure of this code and the large computation granularity inherent in the problem instances. The observed performance improvement depends on the load balance that results from a heterogeneous data domain decomposition. The performance benefit obtained by using heterogeneous processors is offset when the data domain is evenly distributed across the workers as shown in Table 7.21. As we have observed in the other codes the effective parallelism is diminished by load imbalance and a single cluster of SGI's would have been a better choice for these problem instances.

Problem	**Configuration**			**PDUs**			**Elapsed Time (msec)**	**% Increase in Etime with respect to balanced load**
	C_1	C_2	C_3	A_1	A_2	A_3		
FA-2	6	5	0	400	400	0	3297	101%
SW-2	6	8	0	44	44	0	23573	103%
SW-3	6	8	8	65	65	65	42996	89%

Table 7.21: Benefit of heterogeneous data domain decomposition for CL

The impact of endian conversion on CL was minimal due to compute-bound nature of the computation, see Table 7.22. The CL workers at the leaves convert their data in parallel before sending it up the tree and out to the recorder object. The data is a simple record of a few integers that reflects a score for the current source sequence as compared with the target sequences stored with each worker. Not surprisingly conversion has an almost neg-

Problem	**Configuration**			**PDUs**			**Etime (sec)**	**T_c (msec/cycle)**		**% increase in T_c**
	C_1	C_2	C_3	A_1	A_2	A_3		predicted	actual	
FA-1	4	0	0	72	0	0	155	100.8	107.9	2%
FA-2	6	5	0	*622	133	0	1656	1154.0	1150.6	1%
SW-1	6	0	0	24	0	0	3673	2438.5	2552.8	1%
SW-2	6	8	0	82	16	0	12048	7514.0	8372.2	4%
SW-3	6	8	8	*170	34	17	22800	15618.8	15844.9	0%

Table 7.22: Impact of endian conversion for CL

ligible impact on CL. Conversion also has no effect on the selection of heterogeneous processors and the elapsed times observed with conversion enabled are still superior to the best single cluster times.

Finally the use of co-scheduling provides performance benefits for CL. The dominant communication topology for CL is a tree. Under a random placement it is likely that children and parents may be placed in different processor clusters with a larger amount of communication crossing the router. We present the results of co-scheduling for CL in Table 7.23. The results shown are for problem instances with co-scheduling disabled.

Problem	**Configuration**			**PDUs**			**Elapsed Time (msec)**	**% Increase in Etime with respect to co-scheduling**
	C_1	C_2	C_3	A_1	A_2	A_3		
FA-2	6	5	0	622	622	622	2130	30%
SW-2	6	8	0	82	16	0	14985	24%
SW-3	6	8	8	170	34	17	29100	28%

Table 7.23: Benefit of co-scheduling for CL

The experimental results obtained for GE, STEN, FEM, and CL support our thesis. Scheduling may be performed automatically, efficiently, and profitably for a range of data parallel computations. The applications in the test suite ranged from tightly- to loosely-coupled, included small- to large-grained problem instances and both floating-point and integer dominated computations. The results also show that the method is accurate and predictable and suffers tolerable runtime overhead. Accuracy of the method is important because it helps validate the simulation results.

Scheduling in a heterogeneous environment was shown to provide a significant performance benefit, but required that partitioning and placement be done carefully. Processor selection and heterogeneous data domain decomposition are critical to effective partitioning and co-scheduling is critical to effective placement. We showed that the use of heterogeneous processors may provide a performance benefit when the computation granularity was sufficiently high and required a proper data domain decomposition. When the data

domain was decomposed evenly across all workers the load imbalance eliminated the benefit of using heterogeneous processors and reduced the effective parallelism. Co-scheduling was needed to reduce communication costs and the benefit was dependent on the communication topology.

We also provided evidence that the primary cost of heterogeneity, endian conversion, may be tolerated in many cases. Proper placement of conversion functions that ensure parallel execution of conversion operations is one way that conversion overhead is kept low.

The results indicate that the precise costs or benefits experienced are problem and environment dependent. Different problem sizes may exhibit different performance behavior due to memory and cache effects. For very large problems it is possible that paging also had an impact. However the suite of codes and problem instances were varied enough to suggest several trends in the experimental results. Prophet overhead and the cost of endian conversion is on the order of a few milliseconds for all codes. The benefit of heterogeneous processors over the single fastest cluster (SGI's) ranged from 10-40% with a much higher benefit over the two slower clusters (Sparc2's and IPC's). This benefit was generally higher for problem instances with larger computation granularity. The benefit of a heterogeneous data domain decomposition was close to 100% for large problems and between 10-30% for smaller problems. The benefit of co-scheduling ranged from 30-75%, but will depend on the communication topology and the computation granularity. For example, STEN is a fairly tightly-coupled code that benefits a great deal by co-scheduling, around 75%, while CL is more loosely-coupled and the benefits are more modest, closer to 30%.

Chapter 8 Summary and Future Work

We have studied the problem of scheduling data parallel computations in heterogeneous computing environments. A scheduling framework was developed to study the scheduling problem in local- to wide-area network environments. An implementation of the framework called Prophet was completed and integrated into the Mentat-Legion parallel processing system. The Prophet system was used to confirm our thesis that the scheduling of data parallel computations could be automated efficiently at runtime with a large performance benefit in many instances. The experimental results also showed that the performance benefit obtained by using heterogeneous processors in multiple processor clusters required careful data domain decomposition and task placement.

The general applicability of Prophet was confirmed in simulation by the Prophesy simulator. The simulation results indicated that performance close to optimal can be expected in the vast majority of cases. The simulation results were validated by the experimental results. Prophesy was also used to study the feasibility of wide-area parallel processing over a range of network environments and problem granularities.

In the remainder of this chapter we discuss a number of topics that warrant further investigation beyond this dissertation. These topics fall in two broad areas, extending the framework to explore other dimensions of the scheduling problem, and generalizing the network model to an environment that may be wide-area and highly shared.

8.1 Impact of Resource Sharing

In Chapter 3 we presented a model for resource sharing based on resource reservations. The idea behind this model is that memory, CPU cycles, and communication bandwidth could be reserved in some manner, thus providing a guarantee of availability and some measure of predictability. Increased predictability means that cost prediction would be more accurate, and scheduling would be more effective. This model of resource sharing also has the nice property that dynamic load balancing due to unpredictable resource sharing is unnecessary. In some systems a resource reservation scheme for certain resources may be feasible.

However, the more general case is a shared system that can offer limited guarantees on resource availability. One solution to this problem is to avoid using resources that are heavily used by other users and hope that these resources remain mostly unused. We have adopted a variant of this simple solution via a load threshold in our implementation. Since resource usage in the recent past is a good indicator of near-term future usage, this strategy is not as naive as it seems. However this strategy would limit the available resources that we could use in general.

Sharing introduces two problems, static cost prediction and dynamic load balance. Static cost prediction must reflect the sharing of system resources. The impact of reduced memory, CPU cycles, and communication bandwidth must be factored in to the cost equations. It is clear that the impact of sharing will be negative when compared with a dedicated set of resources. Research is needed to quantify this impact.

Dynamic load balancing is needed when the degree of sharing varies widely during the course of program execution. A mechanism is needed to detect that resource usage has changed at runtime and to adjust the schedule to accommodate these changes. For example, a highly loaded processor may have work shifted to a lightly loaded processor as in [62]. In the extreme case we might even retract a processor from the active set that are working on the computation as in [13]. Dynamic load balancing may also be needed if the

problem is irregular and the workload distribution unpredictable. We could rerun the partitioning and placement algorithm in these cases. However this is a global strategy that is not scalable. More scalable dynamic load balancing strategies are given in [47]. An important part of dynamic load balancing is to determine when it is beneficial to perform the load rebalancing. We could extend the callback mechanism to add additional information that would be useful in making this decision. A callback such as *cycles_left* could return the iterations remaining, if it is known. This could be used to estimate the amount of time remaining in the computation and help Prophet decide if dynamic load balancing is worthwhile.

8.2 Functional Parallelism

This thesis has explored one dimension of the scheduling problem — data parallel computations on workstations and multicomputers. A class of computations that exhibit coarse-grain heterogeneity or embedded parallelism may be suitable for the metasystem environment. These computations contain functional or task parallelism that may reflect different resource affinities. Computations such as the Darpa Image Understanding Benchmark [90] and the Multidisciplinary Optimization Problems (MDO) identified by Nasa are examples. There is an opportunity for exploiting resource heterogeneity by matching the tasks to the resources that we predict to deliver the best performance. We have done this already with data parallel computations via T_c.

Scheduling functional parallel computations will require additional user or compiler support to provide affinity information. For example, if a task is vectorizeable this information must be made available at runtime. A technique known as analytic benchmarking [26] has been proposed as a means of gathering this information — the codes are benchmarked on all possible machine configurations and problem sizes, and an affinity matrix is formed. This is a very tedious process and a more viable strategy is needed.

A related topic is to extend the Prophet implementation to a more general metasystem environment containing different machine classes. In this environment parallel computations may have different implementations. For example, we may want to have different source code implementations of a image convolution computation based on whether it is run on a multiprocessor, multicomputer, networks of workstations, or vector machine. This is known as implementation families [3] and it would fit in nicely with our model — a set of callbacks would be provided for *each* implementation. Implementation families would likely contain highly tuned and optimized implementations.

One difficulty with multiple implementations is the issue of compatibility. It may not make sense to decompose a single problem across both a vector and MPP machine because the implementations are incompatible. For example, the implementations may decompose the data domain differently. Some implementations are incompatible because it is not possible to perform accurate format conversions between the machines. We view compatibility as a constraint that must be expressed to the system via a callback. Other constraints may include restrictions on the number of processors. For example, some scientific applications require a number of processors that is even, odd, or a power of two. Additional constraints may include memory demands. This could be specified via a *memory* callback that returns the memory demands for a particular implementation.

8.3 Wide-area Parallel Processing

The results obtained by Prophesy indicate that wide-area parallel processing may be feasible for large-grained computations. The difficulty is that as the network becomes more wide-area with current internet technology, the ability to estimate costs becomes more difficult and predictability begins to decrease rapidly. The degree of bandwidth sharing and number of router hops makes communication delays highly unpredictable. However the spread of on-line wide-area gigabit networks promises to deliver more bandwidth and perhaps greater predictability due to a reduced number of routing hops.

Another difficulty is a scalable and accurate resource availability mechanism. Long latencies in wide-area networks means that load information may become stale quite rapidly. Low latency communication is essential for updating state information. However if resources are dedicated, then this problem becomes less severe. Another solution is provided by the site-based model discussed in Section 3.1.1, where resource information is kept local and the scheduling request is propagated across the sites.

Additional costs such as I/O and data distribution need to be considered in this environment. For example, a site with slower machines but with direct access to the disk where the data domain is stored might be better than a remote site with faster resources. In this case we move to computation to the data instead of moving the data to the computation. This can be modelled by using the $T_{startup}$ term in (Eq.4.8). Additional information that reflects the cost of getting data from the local disk and transmitting it to a remote site will be needed.

In general experimentation with computations running wide-area is needed to get a handle on the cost variance in this setting. An important issue is whether better performance can be expected using wide-area resources over using local resources even in the face of unpredictability.

8.4 Multiprogramming

Another dimension of the scheduling problem is support for job scheduling or multiprogramming. This thesis has studied the scheduling of a single job or computation with elapsed time as the sole metric. In a shared environment higher level scheduling policies are needed to provide some level of system throughput. The problem is complicated by the fact that we may have both parallel computations and sequential jobs to schedule together. We want to keep throughput high but not at the expense of the parallel computations.

Traditional multiprogramming techniques such as time-slicing will work well for independent jobs, but less well for parallel computations in which related tasks ought to be scheduled together. Ideally, we would want like to gang schedule the parallel computations, and time-slice the others. Research into hybrid scheduling policies and production workload studies are needed. This work will be based on exploiting information about the jobs and the environment. This adds another dimension of heterogeneity — sequential and parallel jobs.

8.5 Compiler Support

This thesis has demonstrated that much of the scheduling process may be automated for the programmer. However in the Mentat-Legion implementation of the framework the programmer is responsible for the final stage of scheduling, instantiation, and providing the task implementation. Compiler technology with language support can be used to automate this process for regular data parallel computations based on *1-D* and *2-D* structures [41][57]. Data parallel language extensions to Mentat-Legion are being developed together with the supporting compilation technology [94]. Automatically generating some of the callbacks also looks promising. For example, the language supports a notion of subset parallelism which corresponds to the *PDU* and provides communication topology information. Information about the data relationships is also provided which can be used to support automatic data decomposition.

The compiler may also be able to automatically generate the necessary conversion calls needed to accommodate heterogeneous data formats among machines. We describe a strategy for automating conversions in [31]. The compiler can also exploit knowledge of the communication topology to insert the conversion functions in a way that reduces the impact of the conversion overhead.

Finally a combination of language annotations and compiler support is a possible direction for functional parallel computations. For example, the compiler should be able to generate a callback such as *affinity* that will return any machine affinities.

References

[1] F.D. Anger, J. Hwang, and Y. Chow, “Scheduling with Sufficient Loosely Coupled Processors,” *Journal of Parallel and Distributed Computing*, Vol. 9, 1990.

[2] J.B. Armstrong, D.W. Watson, and H.J. Siegel, “Software Issues for the PASM Parallel Processing System,” in *Software for Parallel Computation*, J.S. Kowalik, ed., Springer-Verlag, Berlin, 1993.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy, “Distribution and Abstract Types in Emerald,” University of Washington, TR 85-08-05, August, 1985.

[4] E.A. Arnould et al, “The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers,” *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[5] M.J. Atallah et al, “Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations,” *Journal of Parallel and Distributed Computing*, Vol. 16, 1992.

[6] L. Bergman et al, “CASA Gigabit Testbed: 1993 Annual Report,” Technical Report CCSF-33, Caltech Concurrent Supercomputing Facilities, Pasadena, CA, May 1993.

[7] F. Berman, and B. Stramm, “Communication-Sensitive Heuristics and Algorithms for Mapping Compilers,” *Sigplan PPEALS 1988*, July 1988.

[8] B.N. Bershad et al, “A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems,” *IEEE Transactions on Software Engineering*, SE-13, 1987.

[9] S.H. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publishers, 1987.

[10] N.S. Bowen, C.N. Nikolau, and A. Ghafoor, “On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computing Systems,” *IEEE Transactions on Computers*, Vol. 41, March 1992.

[11] R. Butler, and E. Lusk, “Monitors, messages, and clusters: The p4 parallel programming system,” *Parallel Computing*, Vol. 20, 1994.

[12] N. Carriero, “Linda in Heterogeneous Computing Environments,” *International Parallel Processing Systems IPPS*, 1992.

[13] N. Carriero et al, “Adaptive Parallelism with Piranha,“ Technical Report 954, Yale University, 1993.

[14] T.L. Casavant, and J.G. Kuhl, “A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems,” *IEEE Transactions on Software Engineering*, Vol. 14, February, 1988.

[15] S. Chen et al, “A Selection Theory and Methodology for Heterogeneous Supercomputing,” *Workshop on Heterogeneous Processing IPPS*, April 1993.

[16] A.L. Cheung, and A.P. Reeves, “High Performance Computing on a Cluster of Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, September 1992.

[17] R. Cytron, “Useful Parallelism in a Multiprocessing Environment,” *Proceedings of the 1985 International Conference on Parallel Processing*, 1985.

[18] H.G. Dietz, W.E. Cohen, and B.K. Grant, “Would you run it here... or there? AHS: Automatic heterogeneous supercomputing,” *Proceedings of the 1993 International Conference on Parallel Processing*, 1993.

[19] Digital Equipment Corporation, *Digital’s GIGAswitch Platform*, 1992.

[20] V. Donaldson, F. Berman, and R. Paturi, “Program Speedup in a Heterogeneous Computing Network,” *Journal of Parallel and Distributed Computing*, Vol. 21(3), 1994.

[21] D.L. Eager, E. D. Lazowska, and J. Zahorjan, “Adaptive Load Sharing in Homogeneous Distributed Systems, ” *IEEE Transactions on Software Engineering*, Vol. 12, May 1986.

[22] H. El-Rewini and T.G. Lewis, “Scheduling parallel program tasks onto arbitrary target machines,” *Journal of Parallel and Distributed Computing*, Vol. 9, 1990.

[23] M.M. Eshaghian, “Cluster-M Parallel Programming Model,” *International Parallel Processing Systems IPPS*, 1992.

[24] D. Forslund, “Recent results on high speed networking and distributed computing in the Advanced Computing Laboratory,” *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop* 1991.

[25] G. Fox et al, *Solving Problems on Concurrent Processors Volume I*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[26] R. Freund, “Optimal Selection Theory for Superconcurrency,” *Supercomputing 1989*, 1989.

[27] F. Freund and H.J. Siegel, “Heterogeneous Processing,” *IEEE Computer*, June 1993.

[28] G.H. Golub and J.M. Ortega, *Scientific Computing and Differential Equations*, Academic Press, Inc., 1992.

[29] A.S. Grimshaw, J.B. Weissman, and E.A. West, “UVa Experiences with the Mentat MetaSystems Testbed,” *Workshop on Cluster Computing*, 1992.

[30] A.S. Grimshaw, E. A. West, and W.R. Pearson, “No Pain and Gain! - Experiences with Mentat on Biological Application,” *Concurrency: Practice & Experience*, Vol. 5(4), June 1993.

[31] A.S. Grimshaw, J.B. Weissman, E.A. West, and E. Loyot, “Metasystems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems,” *Journal of Parallel and Distributed Computing*, Vol. 21(3), June 1994.

[32] A.S. Grimshaw, J. B. Weissman, and W. T. Strayer, ‘‘Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,’’ to appear in *ACM Transactions on Computer Systems*.

[33] A.S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, May 1993.

[34] A.S. Grimshaw, W.A. Wulf, J.C. French, A.C. Weaver, and P.F. Reynolds Jr., “Legion: The Next Logical Step Toward a Nationwide Virtual Computer,” Computer Science Technical Report, University of Virginia, CS 94-21, June, 1994.

[35] A.S. Grimshaw, A. Nguyen-Tuong, and W.A. Wulf,” Campus-Wide Computing: Early Results Using Legion at the University of Virginia, CS-95-19, March 1995.

[36] A.S. Grimshaw, “The Mentat Run-Time System: Support for Medium Grain Parallel Computation,” *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[37] A.S. Grimshaw and V. E. Vivas, “FALCON: A Distributed Scheduler for MIMD Architectures”, *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, Atlanta, GA, 1991.

[38] A.S. Grimshaw, D. Mack, and T. Strayer,‘‘MMPS: Portable Message Passing Support for Parallel Computing,*’’ Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[39] A. Gupta, and A. Tucker, “Exploiting Variable Grain Parallelism at Runtime,” *Sigplan PPEALS 1988*, July 1988.

[40] R.V. Hanxleden, and L.R. Scott, “Load Balancing on Message Passing Architectures,” *Journal of Parallel and Distributed Computing*, Vol. 13, 1991.

[41] P. J. Hatcher, et al, “Data-Parallel Programming on MIMD Computers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 377-383.

[42] P.T. Homer and R.D. Schlichting, “A Software Platform for Constructing Scientific Applications from Heterogeneous Resources,” *Journal of Parallel and Distributed Computing*, Vol. 21(3), June 1994.

[43] C. Huang and P.K. McKinley, “Communication Issues in Parallel Computing Across ATM Networks,” *IEEE Transactions on Parallel and Distributed Technology*, Vol. 2(4), 1994.

[44] M.A. Iqbal, “Partitioning Problems in Heterogeneous Computer Systems,” *Workshop on Heterogeneous Processing IPPS*, April 1993.

[45] A. Khokhar et al, “Heterogeneous Supercomputing: Problems and Issues,” *Workshop on Heterogeneous Processing IPPS*, April 1992.

[46] A. Khokhar et al, “Heterogeneous Computing: Challenges and Opportunities,” *IEEE Computer*, June 1993.

[47] V. Kumar, A.Y. Grama, and V.N. Rao, “Scalable Load Balancing Techniques for Parallel Computers,” *Journal of Parallel and Distributed Computing*, Vol. 22, July 1994.

[48] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan-Kaufmann Publishers, 1992.

[49] J. Li and H. Kameda, “Optimal Static Load Balancing in Star Network Configurations with Two-Way Traffic,” *Journal of Parallel and Distributed Computing*, Vol. 23, 1994.

[50] P.C. Liewer, et al, “Dynamic Load Balancing in a Concurrent Plasma PIC Code on the JPL/Caltech Mark III Hypercube,” *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.

[51] W.B. Ligon III and U. Ramachandran, “Evaluating Multigauge Architectures for Computer Vision,” *Journal of Parallel and Distributed Computing*, Vol. 21(3), June 1994.

[52] M.J. Litzkow et al, “Condor - a hunter of idle workstations,” In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[53] J. Liu, and V.A. Saletore, “Self-Scheduling on Distributed-Memory Machines,

Proceedings Supercomputing 1993.

[54] V.M. Lo, "Algorithms for Static Task Assignment and Symmetric Contraction in Distributed Computing Systems," *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[55] V.M. Lo, "Temporal Communication Graphs: Lamport's Process-Time Graphs Augmented for the Purpose of Mapping and Scheduling," *Journal of Parallel and Distributed Computing*, Vol. 16, 1992.

[56] V.M. Lo et al, "OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures," CIS-TR-89-18a, Department of Computer Science, University of Oregon, April 1992.

[57] D.B. Loveman, High Performance Fortran," *IEEE Transactions on Parallel and Distributed Technology: Systems and Applications*, Vol. 1(1), February 1993.

[58] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," *ACM Sigplan Conference on Programming Languages*, 1992.

[59] C. McCreary and H. Gill, "Efficient Exploitation of Concurrency using Graph Decomposition," *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[60] C.R. Mechoso, J.D. Farrara, and J.A. Spahr, "Running a Climate Model in a Heterogeneous Distributed Computer Environment," *Proceedings of the Third International IEEE Symposium on High Performance Distributed Computing*, August 1994.

[61] R. Mirchandaney, D. Towsley, and J.A. Stankovic, "Adaptive Load Sharing in Heterogeneous Distributed Systems," *Journal of Parallel and Distributed Computing*, Vol. 9, 1990.

[62] N. Nedeljkovic, and M.J. Quinn, "Data-Parallel Programming on a Network of Heterogeneous Workstations," *Proceedings of the First Symposium on High-Performance Distributed Computing*, Sept. 1992.

[63] H. Nicholas et al, "Distributing the comparison of DNA and protein sequences across heterogeneous supercomputers," *Proceedings Supercomputing* 1991, November 1991.

[64] D.M. Nicol, and F.H. Willard, "Problem Size, Parallel Architecture, and Optimal Speedup," *Journal of Parallel and Distributed Computing*, Vol. 5, 1988.

[65] D.M. Nicol and D.R. O'Hallaron, "Improved Algorithms for Mapping Pipelined and Parallel Computations," *IEEE Transactions on Computers*, Vol. 40(3), 1991.

[66] D.M. Nicol and P.F. Reynolds, Jr., "Optimal Dynamic Remapping of Data Parallel Computations," *IEEE Transactions on Computers*, Vol. 39(2), 1990.

[67] D. Notkin et al, "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accomodating Heterogeneity," *CACM*, Vol. 30(2), February 1987.

[68] D. Notkin et al, "Interconnecting Heterogeneous Computer Systems," *CACM*, Vol. 31(3), March 1988.

[69] C. Polychronopoulis and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, Vol. c-36(12), December 1987.

[70] M.J. Quinn, “Parallel computing: theory and practice”, 2nd ed, McGraw-Hill, 1994.

[71] D.A. Reed, L.M. Adams, and M.L. Patrick, “Stencils and Problem Partitioning: Their Influence on the Performance of Multiple Processor Systems,” *IEEE Transactions on Computers*, Vol. c-37(7), July 1987.

[72] D.A. Reed, and R.M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, 1987.

[73] L. Revor, *DQS Users Guide*, Computing and Telecommunications Division, Argonne National Laboratory, September 1992.

[74] V. Sarkar, “Determining Average Program Execution Times and their Variance,” *Sigplan Programming Language Design and Implementation*, 1989.

[75] V. Sarkar, and J. Hennessy, “Compile-time Partitioning and Scheduling of Parallel Programs,” *Sigplan Notices ‘86 Symposium on Compiler Construction*, 1986.

[76] W. Shu and L.V. Kale, “Chare Kernel — a Runtime Support System for Parallel Computations,” *Journal of Parallel and Distributed Computing*, Vol. 11, 1991.

[77] W. Shu and L.V. Kale, “A Dynamic Scheduling Straategy for the Chare-Kernel System,” *Proceedings Supercomputing* 1989.

[78] B.S. Siegell and P. Steenkiste, “Automatic Generation of Parallel Programs with Dynamic Load Balancing,” *Proceedings of the Third International IEEE Symposium on High Performance Distributed Computing*, August 1994.

[79] H.S. Stone, *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, 1987.

[80] H.S. Stone, “Multiprocessor Scheduling with the Aid of Network Flow Algorithms,” *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977.

[81] M.J. Strohl, “High Performance Distributed Computing in FDDI Networks,” *IEEE LTS*, Vol. 2, May 1991.

[82] Sun Microsystems Inc., *Network Programming Guide—External Data Representation Standard: Protocol Specification*, 1990.

[83] V.S. Sunderam, “PVM: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, Vol. 2(4), December, 1990.

[84] P. Tang and P.C. Yew, “Processor Self-Scheduling for Multiple Nested Parallel Loops,” *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[85] C.A. Thekkath, H.M. Levy, and E.D. Lazowska, “Efficient Support for Multicomputing on ATM Networks,” Technical Report 93-04-03, 1993.

[86] D.F. Towsley, “Allocating programs containing branches and loop within a multiple processor system,” *IEEE Transactions on Software Engineering*, Vol. SE-2, October 1986.

[87] J. Ullman, “NP-complete scheduling problems,” *Journal of Computing System Science*, Vol. 10, 1975.

[88] M. Wang et al, “Augmenting the Optimal Selection Theory for Superconcurrency,” *Workshop on Heterogeneous Processing IPPS*, April 1992.

[89] D.W. Watson et al, “A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments,” *Journal of Parallel and Distributed Computing*, Vol. 21, No. 3, 1994.

[90] C.C. Weems et al, “The DARPA image understanding benchmark for parallel computers,” *Journal of Parallel and Distributed Computing*, Vol. 11(1), 1991.

[91] J.B. Weissman and A.S. Grimshaw, “A Framework for Partitioning Parallel Computations in Heterogeneous Environments,” to appear in *Concurrency: Practice and Experience*, Vol. 7(5), August 1995.

[92] J.B. Weissman, A.S. Grimshaw, and R. Ferraro, “Parallel Object-Oriented Computation Applied to a Finite Element Problem,” *Journal of Scientific Programming*, Vol. 2(4), 1993.

[93] J.B. Weissman and A.S. Grimshaw, “Network Partitioning of Data Parallel Computations,“ *Proceedings of the Third International IEEE Symposium on High Performance Distributed Computing*, August 1994.

[94] E.A. West, and A.S. Grimshaw, “Braid: Integrating Task and Data Parallelism,” *Proceedings of Frontiers of Massively Parallel Processing*, 1995.

[95] R.D. Williams, “Performance of dynamic load balancing algorithms for unstructured mesh calculations,” *Concurrency: Practice and Experience*, Vol. 3(5), October 1991.

[96] D.B. Wortman, S. Zhou, and S. Fink, “Automating Data Conversion for Heterogeneous Distributed Shared Memory,” *Software: Practice and Experience*, Vol. 24(1), January 1994.

[97] T. Yang and A. Gerasoulis, “A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors,” *Scalable High Performance Computing Conference S*HPCC-92, 1992.

[98] S. Zhou et al, “Utopia: A Load Sharing Facility for Large Heterogeneous Distributed Computer Systems,” *Software: Practice and Experience*, Vol. 23(12), December 1993.

[99] S. Zhou et al, “Heterogeneous Distributed Shared Memory,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3(5), September 1992.