LOAD BALANCING TECHNIQUES FOR DISTRIBUTED
PROCESSING ENVIRONMENTS

The members of the Committee approve the doctoral dissertation of Daniel Joseph Harvey

Sajal Das Supervising Professor	______________________
Rupak Biswas	______________________
Dianne Cook	______________________
Mohan Kumar	______________________
Behrooz Shirazi	______________________
Bob Weems	______________________
Dean of the Graduate School	______________________

LOAD BALANCING TECHNIQUES FOR DISTRIBUTED PROCESSING ENVIRONMENTS

by

Daniel Joseph Harvey

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
August 2001

ACKNOWLEDGMENTS

First I want to thank my principal adviser, Dr. Sajal Das, whose encouragement is a principal reason for my finishing this dissertation. Many times, when I was at the point of giving up, Dr. Das encouraged me to continue. Dr. Das was also instrumental in pushing me towards many successful publication efforts. I am sure that this experience will be vital as I continue to pursue research interests. I also want to express my deepest appreciation to Dr. Rupak Biswas of NASA Ames Research Center for his help. He always seemed to have time to critique my work and give the excellent pointers as to how it could be improved. His extensive knowledge and insight allowed him to consistently focus on areas that needed clarification. The amount of time he dedicated was well beyond any expectations that I could have had. I feel very fortunate to have had the privilege of working with both Dr. Das and Dr. Biswas and I especially value the friendship that we have developed over the years.

I'm also appreciative to Dr. Lenny Oliker or Lawrence Berkeley National Laboratories who took the time to review my work and with whom I was able to co-author some papers. His insights were very helpful.

I also would like to express appreciation to Drs. Dianne Cook, Mohan Kumar, Behrooz Shirazi and Bob Weems for being willing to serve on my committee. A special note of thanks goes to Dr. Weems for helping me work through the administrative details relating to graduation.

Most of all I would like to express heartfelt appreciation to my family who exhibited tremendous patience as I worked towards completing my dissertation. There were many times that they had to endure great sacrifice as I worked two jobs and at the same time worked towards completing my education. I am also grateful that God has given me the ability to study, to learn, and to pursue research in computer science.

Finally, I am grateful to NASA Ames Research Center (Moffet Field, California,) for supporting this work during my visit in the summer of 1999 and for making available multicomputer resources which were used to conduct experiments. Without their cooperative agreements (NCC-2-5395, and NAS 2-14303) this work would not have been possible. The State of Texas should also be acknowledged for their Advanced Research Grant Number TARP-97-003594-013.

June 26, 2001

ABSTRACT

LOAD BALANCING TECHNIQUES FOR DISTRIBUTED PROCESSING ENVIRONMENTS

Publication No. ________

Daniel Joseph Harvey, Ph.D.

The University of Texas at Arlington, 2001

Supervising Professor: Sajal Das

Improvements in computer technology over the past several decades have led to great interest in utilizing distributed multicomputer systems to solve complex problems that would be otherwise impractical or inefficient using a single processor. To achieve optimized multicomputer performance, however, it is essential to evenly distribute the workload among the available processors. For applications such as dynamic mesh adaptation of unstructured grids, achieving a balanced load is difficult because of the rapidly changing nature of the grid.

In this dissertation, the load balancing problem is addressed in several ways. First, three topology independent dynamic load balancing schemes are proposed that are directly implementable on a variety of multicomputer architectures including the hypercube. Experiments using synthetically generated loads compare these algorithms to other popular load balancers and demonstrate that the proposed algorithms compare favorably. Next, the approach is modified to provide a global view of system load so that the dynamic schemes can also be applied to unstructured mesh applications where dynamic

load balancers are not often used. Simulation experiments using workload data obtained from actual solvers demonstrate that the proposed algorithms are an effective alternative.

Finally, a novel partitioner, called MinEX, is presented that is specifically designed to execute in a distributed computing environment. Unlike dynamic load balancers, this approach partitions processor workloads prior to execution. As the load balance shifts, application processing is temporarily suspended and repartitioning takes place. The MinEX partitioner has the novel objective to minimize runtime rather than to achieve an equal workload on each processor. This new approach takes into account the cost of inter-processor communication, as well as the cost of moving data sets between processors, and accounts for latency tolerance. MinEX is evaluated by comparing its performance in two ways. First results are obtained using data produced by unstructured mesh solvers. Next, a solver for the classic N-body application is developed so that direct comparisons between MinEX and MeTiS (a state-of-the-art partitioner) can be made. Results conclude that MinEX achieves superior performance in a truly distributed computing environment with lower bandwidths and slower interconnects.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

CHAPTER 1

INTRODUCTION

In the last couple of decades, technological improvements have greatly reduced the cost of parallel and distributed computing systems. These improvements led to great interest in developing *scheduling* algorithms which guarantee efficient use of available resources. Operating systems have long dealt with issues of local scheduling where time-slices are allocated to processes executing on a single CPU. Distributed operating systems must now also deal with global scheduling problems such as allocating processes to particular processors and possibly moving processes from one CPU to another during execution. Responsibility for scheduling can be centrally controlled by a single CPU or responsibility can be distributed throughout the multiprocessor system. Scheduling solutions can be either static or dynamic. In *static* scheduling algorithms, the decisions are made prior to execution time when resource requirements are estimated. For example, application requirements might be predicted based upon previous runs. On the other hand, *dynamic* scheduling algorithms allocate/reallocate resources at runtime.

Closely related to the scheduling problem is *load balancing*. In this case, the application shifts workload among processors in accordance with system requirements. Load balancing algorithms, like scheduling, can also be dynamic or static. In static load balancing algorithms [64, 67], the decisions are made at compilation time when resource requirements are estimated. On the other hand, dynamic algorithms [15, 25, 35] allocate/reallocate resources at runtime based on a set of system parameters, which may determine when jobs can be migrated and also account for the associated overhead in such a transfer [66]. Determining the parameters to be maintained and how to broadcast them among processors are important design considerations, normally resolved by distributed scheduling policies [28, 52].

A *semi-dynamic* load balancing method, which is particularly useful for processing unstructured adaptive meshes [6, 20, 60], makes use of a partitioner. Using this approach, the system workload is balanced through a partitioning algorithm before execution commences. When the load eventually becomes uneven due to runtime adaptations, execution is temporarily suspended and the load is re-partitioned and re-balanced. To be effective, the semi-dynamic approach should take into account the cost of moving data sets after partitioning and the cost of inter-processor communication that is incurred when execution continues.

1.1 Contributions of this Dissertation

The purpose of this research effort is to introduce novel dynamic and semi-dynamic load balancing approaches. The first part of the research introduces three dynamic load balancing algorithms that utilize a topology independent structure known as a Symmetric Broadcast Network (SBN). These algorithms are analyzed mathematically and also compared experimentally to several other popular dynamic algorithms with favorable results. All three of the algorithms achieve excellent load balance, while a heuristic version minimizes the communication overhead associated with the balancing process. One of these algorithms demonstrates the versatility of the approach by a direct hypercube embedding. Finally, the SBN approach is applied to simulate processing of an unstructured mesh adaptation application. Mesh adaptation applications traditionally utilize a semi-dynamic partitioning approach and do not often employ dynamic load balancers because they generally lack of a global system view. Since dynamic load balancers do not stop execution as the load is balanced, The success of the SBN algorithms in connection with this application is an important contribution. SBN's effectiveness is measured by comparing results with a popular semi-dynamic framework, called PLUM [60, 6, 20], which makes use of any widely available partitioner for load balancing of unstructured adaptive meshes. Results show that SBN achieves a lower dataset movement cost; albeit at a somewhat higher communication cost. In applications where dataset movement dominates, the SBN approach would be preferred.

The next load balancing approach that is introduced in this dissertation is a partitioner, called MinEX, that is specifically designed to run in a distributed environment. Its primary partitioning goal is to minimize runtime; accomplished by integrating the required data movement into the partitioning process and accounting for the distributed computing configuration. The MinEX partitioner also considers latency tolerance factors through a well defined application interface.

These objectives of MinEX are significantly different from most general purpose partitioners such as MeTiS [44] which create partitions by equally balancing the processing requirements among the processors while keeping communication costs under a user-specified threshold. General purpose partitioners also give no consideration to data re-mapping costs or to latency tolerance. For these reasons, the application frameworks for processing unstructured adaptive meshes, (e.g., PLUM,) breaks up the load balancing operation into two distinct steps. In the first step, the partitioner is called; while in the second step, a re-mapping algorithm is invoked to achieve load balance. Comparisons with results gathered from PLUM and SBN-based experiments, show that the MinEX partitioner achieves a good tradeoff between the cost of communication and data redistribution.

To complement these results, a solver for the N-body application [5] is developed so that direct comparisons between MinEX and a state-of-the-art partitioner, called

MeTiS [44, 45, 46], can be made. Results conclude that MinEX achieves superior performance in a distributed computing environment. Specifically, runtimes achieved in a distributed configuration using the MinEX partitioner are one third of those obtained when MeTiS is utilized.

1.2 Dissertation outline

This section outlines the organization of the chapters of this dissertation. Chapter 2 describes background information needed in subsequent chapters. Chapter 3 defines Symmetric Broadcast Networks (SBNs) and describes the three SBN-based dynamic load balancing algorithms. In chapter 4, the SBN approach is adapted to provide a global system view and establish its viability as an effective alternative to semi-dynamic schemes for processing adaptive mesh applications. Chapter 5 further refines the SBN Basic algorithm and presents experimental results comparing SBN to the PLUM semi-dynamic framework. The MinEX partitioner is described in chapter 6 and MinEX related experiments are presented and analyzed in chapter 7. Conclusions along with areas for future research are discussed in chapter 8.

CHAPTER 2

PRELIMINARIES

This chapter describes preliminary concepts that relate to distributed computing. For example, categories of distributed architectures are described in section 2.1. Middleware packages that are used in distributed environments are discussed in section 2.2.

2.1 Distributed computing architectures

The load balancing methods proposed in this dissertation are implemented on either an IBM SP2 or an SGI Origin2000 and also run on low cost remotely connected clusters of processors. The SP2 belongs to the class of what is commonly known as distributed-memory multicomputers, whereas the Origin2000 is a distributed shared-memory machine. The architectural features of these configurations are briefly discussed in sections 2.1.1 and 2.1.2 for the sake of completeness.

2.1.1 Distributed-memory computers

In a distributed-memory or message-passing architecture, each processor has its own local memory which only it can access directly. Since there is no common bus, remote memory is not directly addressable and data are shared by explicitly sending and receiving messages. In this message-passing model, data transfers require operations to be performed by both the sending and the receiving processors. Such architectures are inherently scalable because they do not require tight coupling from the system to maintain address space or data coherence. A key performance advantage of these machines is the ability for the programmer to explicitly associate specific data with processes. This allows the system to effectively use caches and memory hierarchy on each processing unit. Unfortunately, writing message-passing code is a complex task since the user must explicitly partition the data among the local processors. Once a data structure is distributed, all sections of the code using that data must be implemented in parallel. Thus, it is very difficult to incrementally parallelize a serial code using this paradigm.

The IBM 9076 Scalable Power Parallel SP2 system is an example of a distributed-memory parallel system [2]. Each node consists of a six-instruction issue superscalar RISC6000 processor, running at 66.7 MHz. The nodes are connected through a high-performance switch called the Vulcan chip. The topology of the switch is an any-to-any packet switched network similar to an Omega network. An advantage of this interconnection mechanism is that the available bandwidth between any pair of communicating nodes remains constant[1] regardless of where in the topology the two nodes lie.

2.1.2 Shared-memory multicomputers

A distributed shared-memory architecture combines physically distributed memory together with hardware which treats the memory as a unified, global logical address space. Thus, all of the physical memory is directly addressable from any processor. These systems have the advantage of efficiently supporting a shared-memory programming model, while preserving scalability. A drawback, however, is the overhead necessary for maintaining address and data coherence. To maintain a single global address space, the system must be tightly coupled at either the software or the hardware level. This overhead can degrade performance with increasing number of processors.

[1] The same does not hold true for direct networks such as rings, meshes, or multidimensional toroids which require an additional performance optimization to consider the number of hops between two communicating nodes.

An example of a distributed shared-memory architecture is the SGI Origin2000, the first commercially available 64-bit cache-coherent nonuniform memory access (CC-NUMA) system [70]. Each CPU is a R10000 four-way superscalar RISC processor running at speed of up to 195 MHz.

A small high-performance switch connects two CPUs, memory, and I/O. This module, called a node, is then connected to other nodes via a hypercube interconnection network. An advantage of this interconnection system is that additional nodes and switches can be added to create larger systems that scale with the number of processors. The memory access time is no longer uniform as in a truly shared-memory architecture, and varies depending on its location. Each processor in the Origin2000 has a private cache; specialized hardware maintains the image of a single shared memory as well as cache coherence.

2.1.3 Clusters of workstations

The supercomputing systems described in sections 2.1.1 and 2.1.2 are designed with specialized high-speed networks. In recent years, however, workstations are often linked with conventional networks, such as Ethernet, FDDI, or ATM to form parallel systems. Networks of workstations, (NOWs), take advantage of the fact that workstations are relatively inexpensive, widely available, and often idle. Additionally, the availability of relatively low-cost high-speed networks makes this approach a good price/performance value for compute-intensive problems. Unfortunately, NOWs can efficiently support fewer parallel applications than specialized massively parallel processors, (MPPs), due to their lower bandwidths and higher communication latencies.

2.2 Distributed computing middleware

This section describes middleware that is utilized for data sharing and application processing in distributed computing environments.

2.2.1 Information Power Grid (IPG)

The Information Power Grid (IPG) [36] is intended to provide a convenient interface to the vast, heterogeneous, and geographically-separated computing resources of NASA and/or other IPG partners. The interface hides details of machine particulars, such as location, size, connectivity, and name, thereby presenting a unified virtual machine to developers. A toolkit for ensuring security, robustness, portability, and transparency is being established by the Globus project [38, 43] which is a joint effort by Argonne National Laboratory and the University of Southern California's Information Sciences Institute.

There are a number of software challenges that must be addressed before actual applications can be run in a distributed fashion on the IPG. One of the most important issues is to develop efficient architecture-independent (i.e., portable) solutions for real-life problems (e.g., a numerical solver) running on various machine platforms connected by the IPG. Another important issue is to define performance metrics for assessing the impact of high-latency communications on running a single instance of the application (solver) on multiple clusters (super-nodes) of machines. This will guide the design of more latency-tolerant implementations in which communication will be hidden under and/or overlapped with computation. Clearly, dynamic load balancing in such a heterogeneous environment will play a central role in the overall performance.

2.3 Message-passing programming

One of the fundamental issues in parallel computing is the choice of programming paradigms. Several paradigms have evolved over the last two decades, which include shared-memory programming, message-passing programming, data parallel programming [50], and multi-threading [12, 40, 72]. Some of these paradigms have turned out to be architecture-independent in the sense that parallel programs, software, and tools developed with their help can be easily and efficiently ported onto various machine platforms. In this dissertation, the *message-passing programming* model is used.

Message passing is a popular paradigm for a wide range of parallel systems since it offers programmers direct access to communication between processes on different machines. With the explicit message-passing model, processes in a parallel application have their own private address spaces and share data via explicit messages. Several vendors have demonstrated a message-passing library that is efficiently implemented and portable. However, because of the need for explicit communication, message passing presents a complex programming interface and is difficult to debug. Furthermore, the performance of an application depends heavily on the expertise of the programmer. Programs expressed in this manner may run on distributed-memory multiprocessors, distributed shared-memory machines (CC-NUMA), or a heterogeneous network of workstations. Message Passing Interface (MPI) [58] and Parallel Virtual Machine (PVM) [37] are the two most popular standards for writing message-passing programs, and were successfully ported onto a large number of the above platforms. Active Messages is a low level message-passing programming model that makes machine-dependent communication resources directly available to the programmer.

Although extensive experiments were conducted in this research with both PVM and MPI, MPI is preferred since it has shown superior results on high-performance architectures. Active Messages was not utilized due to its lack of portability and its low-level characteristics. Therefore, results will be reported based upon experience with using MPI for parallel implementations of load balancing algorithms on different platforms. The easy portability of the parallel programs developed on these machines demonstrate the versatility of the algorithms to be presented.

2.3.1 Message Passing Interface

The MPI (Message Passing Interface) software package is currently the most widely-used standard for writing message-passing programs, with its MPI-2 specifications released recently [58]. A grid-enabled implementation of MPI (MPICH-G) was also developed so that users can run MPI programs across remotely connected clusters of processors using the same commands that would be used on a tightly coupled parallel computer [33, 34]. MPICH-G is used by NASA with the Globus meta-computing toolkit [32] to implement their Information Power Grid (IPG).

The main goals for designing this interface are portability, efficiency, and programmability. MPI provides the syntactic and semantic core of library routines for a distributed-memory communication environment. Providing this high-level standardization allows an efficient implementation of lower-level message-passing functions, and the possibility of hardware support by various vendors. MPI is suitable for use by general MIMD (multiple instruction, multiple data stream) programs, as well as those written in the more restricted style of SPMD (single program, multiple data stream). It is a flexible communication library which can be used in a heterogeneous environment and contains bindings to both C and Fortran77. MPI-2 extends the language bindings to C++ and Fortran90.

MPI provides an interface that allows processes in a parallel program to communicate with one another. In MPI-1, processes cannot be added to or deleted from an application after it has been started. The MPI-2 process model, however, allows for the dynamic creation and termination of processes after an MPI application has started (similar to that allowed by PVM). Various communication primitives are supported. Point-to-point communications allow simple blocking and non-blocking send/receive primitives. Collective communications involve groups of processes, and allow operations such as barrier synchronizations, gather/scatters, and global reductions. MPI-2 also provides one-sided communications. This feature allows one process to specify all communication parameters for both the sending and the receiving sides. This is a powerful construct which allows implementors to take advantage of fast communication mechanisms provided by various platforms, such as coherent or non-coherent shared memory, DMA engines, hardware-supported put/get operations, and communication co-processors.

All of the algorithms proposed in this dissertation assume a message passing environment and use the MPI message passing package for communication between processors.

2.3.2 Parallel Virtual Machine (PVM)

PVM is a software environment, similar to MPI, that allows a set of heterogeneous computers to run as a single multiprocessor system, referred to as a *virtual machine* [37]. This virtual machine consists of a user-defined collection of systems including serial, parallel, and vector computers. All machines in the system must run under UNIX. Once the virtual machine is defined, PVM includes routines to automatically spawn tasks on the individual processors and allows them to communicate and synchronize with each other. Each machine is called a host. One processor is usually designated to run the PVM console (PVMC,) also known as the managing process. All other processors of the virtual machine run the PVM daemon (PVMD). PVMC provides a means for interfacing with the PVM library, and PVMD takes care of the task of message passing.

2.3.3 Active Messages

Active Messages [27] is an asynchronous communication mechanism designed to mitigate the message-passing overhead. The idea is to overlap communication latency with computation by exposing the full hardware capabilities to the programmer. Each Active Messages contains a handler address in addition to the data being transferred.

When messages arrive at their destination, the ongoing computation is interrupted and the handler is executed. The handler quickly extracts the message out of the network, and integrates it into a user-specified memory address with a small amount of work. This mechanism relies on a uniform code image in all communicating nodes, as is commonly used in the SPMD programming model. Active Messages are intended to serve as building blocks for creating higher-level communication libraries. The MPI-2 standard, for example, allows for one-sided communication, and could be built on top of Active Messages. The disadvantage of this asynchronous communication style is increased programming complexity and higher probability of deadlock within the network.

CHAPTER 3

SBN-BASED LOAD BALANCING ALGORITHMS

This chapter deals with decentralized load balancing in distributed-memory multicomputers in which processors are connected by a point-to-point network topology and communicate with one another using message passing. The network is homogeneous and any job can be serviced by any processor; however, jobs cannot be rerouted once execution begins. The length of a processor's local jog queue determines its workload.

In particular, three efficient dynamic load balancing algorithms are proposed which make use of a logical and topology-independent communication pattern among processors, called a *Symmetric Broadcast Network* (SBN,) introduced in [16, 17]. These algorithms will be called (i) Basic SBN algorithm, (ii) Hypercube Variant, and (iii) Heuristic Variant.

SBN-based load balancing can be initiated by any processor that has too many or too few jobs to process, based on certain threshold values. *Load balancing messages* are first broadcast so that the current system load of jobs can be estimated. This is followed by *distribution messages* which reassign jobs and/or processors to minimize the possibility of processors becoming idle. SBN-based load balancing runs concurrently while application processing continues.

A topology-independent logical network such as an SBN, helps provide predictable communication patterns to applications that make use of wide area networks of processors for load balancing and inter-processor communication. An SBN is also effective when implementing message-passing applications for multicomputer systems in a portable manner.

The SBN topology can also be embedded into specific networks if efficiency is an issue. For example, the Hypercube Variant algorithm naturally adapts to a hypercube topology, and is thus used to compare with other existing dynamic load balancing strategies implemented on a hypercube. This also helps to determine whether the measured performance improvements are due to the effects of the network topology or the proposed load balancing schemes themselves.

Based on their operational characteristics, the SBN-based algorithms can be classified (according to [68]) as: (a) *Adaptive*, since the performance adapts to the average number of queued jobs; (b) *Symmetrically Initiated*, since both senders and receivers can initiate load balancing; (c) *Stable*, since the network is not burdened with excessive load balancing traffic; and (d) *Effective*, since system performance does not degrade while balancing workloads.

The performance of the proposed SBN-based algorithms are analyzed by conducting two sets of extensive experiments on a 32-processor SGI Origin2000 machine, using the Message-Passing Interface (MPI) paradigm. A preliminary description of these experiments that were performed on an IBM SP2 is presented in [18]. These experiments use Poisson-distributed synthetic loads and compares the performance with other methods such as Random [25], Gradient [54, 57], Sender Initiated [26], Receiver Initiated [26], Adaptive Contracting [26], and Tree Walking [69], as summarized in section 3.1. The experiments demonstrate that the quality of load balancing achieved by the SBN approach compares favorably with respect to three metrics: (i) total completion time, (ii) message traffic per processor, and (iii) maximum variance in processor idle time. For example, under heavy system loads, the SBN algorithms complete in 6% to 22% less time than other balancing algorithms that are analyzed in this chapter. Idle time is also reduced by over two thirds. Under light system loads, the SBN-based algorithms incur significantly less message traffic as compared to other popular balancing algorithms such as Gradient and Receiver Initiated.

The second set of experiments, described in chapter 4, apply the SBN approach to a dynamic mesh adaptation application with actual workload data obtained from a rotorcraft acoustics experiment. Results demonstrate that the SBN-based load balancing can be effective for this class of applications by providing a global view of the system load and by overlapping the load balancing activity with actual processing.

3.1 Popular dynamic algorithms

A wide variety of dynamic load balancing algorithms are available for improving multiprocessor performance [64, 67]. In this section, the underlying characteristics of some of these algorithms (as defined in the literature) are summarized. These algorithms are used to compare the performance of the SBN-based algorithms.

Random [25]

If the number of jobs queued at a given processor is larger than a certain threshold, the additional jobs generated by the processor are randomly distributed among its neighbors. Although a single distribution message may contain multiple jobs, a particular job cannot be migrated multiple times. In other words, once a job is migrated, it is queued for processing.

Gradient [54, 57]

In this approach, jobs migrate from overloaded to lightly-loaded processors based on a systemwide gradient. Each processor maintains, for each of its immediate neighbors, the minimum number of communication hops to the nearest lightly-loaded processor. Whenever these values change, they are broadcast to all the neighbors. However, because of network dynamics, this is only an approximation to the true system load. Each processor also has a load status flag which, by comparison with system thresholds, determines whether the processor is overloaded, lightly loaded, or moderately loaded. Jobs are routed to the neighbor lying on the path to the nearest lightly-loaded processor, and a job can migrate several times before being finally processed.

Receiver Initiated [26]

In this scheme, load balancing is triggered by a lightly-loaded processor. A processor with a load level below the system threshold, broadcasts a job request message to its neighbors which contains queue length information. Upon receiving this message, each neighbor compares its own queue length to that of the requesting processor. If the local queue size is larger, the neighbor processor replies with a single job. To prevent instability under light system load conditions, the requesting processor waits for a specified amount of time for a reply before initiating another request for jobs. Like the Gradient algorithm, it is possible for a job to be migrated multiple times before being finally processed.

Sender Initiated [26]

This algorithm initiates load balancing when processors become overloaded. To prevent instability under heavy system loads, each processor exchanges load information with its neighbors. More precisely, load values are exchanged when a local job queue size is halved or doubled in length, so the load exchange occurs less frequently than the system load changes. When jobs are generated, they are distributed to lightly-loaded neighbors. Like the Random algorithm, multiple job migration is not allowed.

Adaptive Contracting [26]

In this algorithm, the originating processor distributes bids to all its neighbors when jobs are generated. The neighbors respond to the bid with a message containing the number of jobs in their respective local queues. The originating processor then distributes jobs to those neighbors that have loads smaller than the system threshold. The number of jobs migrated is such that jobs are equally distributed among the originating processor and its lightly-loaded neighbors.

Tree Walking [69]

This algorithm utilizes a binary tree topology. The fixed root processor initiates a load balancing operation when one of the processors becomes idle. Processing is temporarily suspended when load balancing is underway. Load balancing is achieved by first broadcasting a balance message through the network. Processors respond by sending their current load level back towards the root using a global reduction operation. The correct systemwide load level is then accurately broadcast. Finally, jobs are distributed so that every processor has an equal number of jobs.

Despite some similarities between the SBN-based approach and the Tree Walking Algorithm (TWA,) there are several major differences as follows:

- TWA always initiates load balancing from a single root processor. TWA is, therefore, more restrictive and has less potential for being fault tolerant. SBN balancing allows any processor to initiate load balancing.

- Application processing is temporarily suspended when TWA executes. TWA is unable to mask balancing overhead by overlapping processing and load balancing. SBN balancing proceeds concurrently as application processing continues; suspension of application processing is not required.

- TWA is designed to perfectly equalize the job count at every processor leading to more message passing. SBN balancing attempts to minimize the possibility that a particular processor will become idle. A perfectly equalized job count is unnecessary.

- TWA triggers load balancing only when processors become idle. SBN anticipates idle conditions and triggers balancing ahead of time.

- TWA requires communication messages to be broadcast to all processors when balancing the system. The SBN Heuristic algorithm (details given in section 3.3.3) allows balancing to be accomplished without this requirement.

3.2 Basic SBN concepts

This section defines the *Symmetric Broadcast Networks* (SBNs) and describes their properties along with the general characteristics of the proposed load balancing algorithms that are based on SBNs.

3.2.1 Definition of SBN

An SBN defines a communication pattern (logical or physical) among the P processors in a multicomputer system [16, 17]. An SBN(d) of dimension $d \geq 0$, is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage. It is constructed recursively as follows:

- A single processor forms the basis network SBN(0) consisting of a single stage, denoted as stage 0, with no communication link.

- For $d > 0$, an SBN(d) is obtained from a pair of SBN($d-1$)s as follows:

  (i) Keep the processor labels in the first SBN($d-1$) unchanged as 0 through $2^{d-1}-1$; relabel the processors of the second SBN($d-1$) as 2^{d-1} through $2^d - 1$.

  (ii) Create an additional communication stage d, containing processors 0 through $2^d - 1$. Connect each processor j in stage $d-1$ to processor $(j + 2^{d-1}) \bmod 2^d$ in stage d.

  (iii) If stage $d-2$ exists, for each processor j in stage $d-2$, define k to be the successor of j in stage $d-1$. Likewise, define m to be the successor of k in stage d (as just created in step (ii) above). Connect processor j in stage $d-2$ to processor m in stage $d-1$.

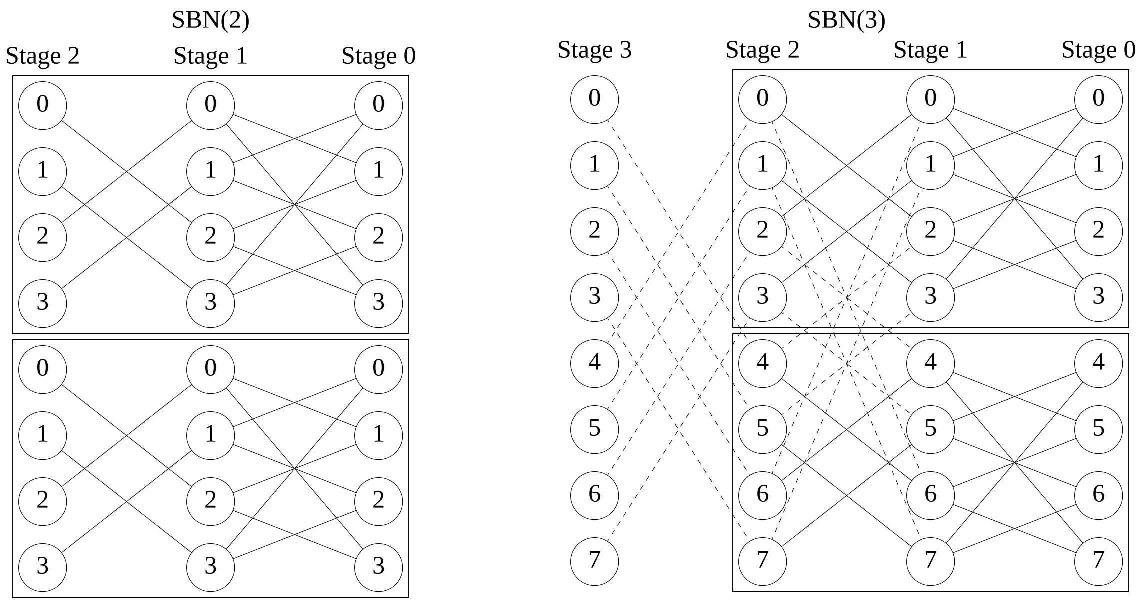


Figure 1. Construction of SBN(3) from a pair of SBN(2)s; Dashes are new connections.

An example of how an SBN(3) is formed from two SBN(2)s is shown in figure 1. An SBN defines unique communication patterns among the processors in the network. For any source processor at stage d of SBN(d), there are $d = \log P$ stages of communication where each processor appears exactly once. The successors and predecessors of each processor in a given stage i are uniquely defined by specifying the label of the originating processor and the communication stage number. Messages originating from source processors are appropriately routed through the SBN.

3.2.2 Properties of SBN

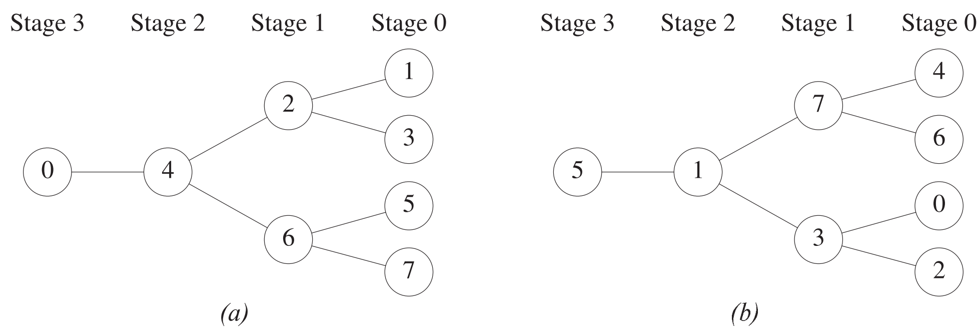


Figure 2. Two communication patterns in SBN(3); (a) processor 0 pattern, (b) processor 5 pattern.

Among a total of eight possible communication patterns in SBN(3), consider the two patterns shown in figure 2. The paths in figure 2(a) are used to route messages originating from processor 0, while those in figure 2(b) are for messages originating from processor 5. Let n_5^s denote a processor label at stage s in figure 2(b) and let n_0^s be the corresponding processor label in figure 2(a). Then $n_5^s = n_0^s \oplus 5$, where $\oplus$ is the exclusive-OR operator. In general, if n_x^s is the corresponding processor in the communication pattern for messages originating from source processor x, then $n_x^s = n_0^s \oplus x$. Thus, all SBN communication patterns can be derived from the template pattern having processor 0 as the root. The predecessor and two successors of n_0^s can then be computed as follows:

Predecessor: $((n_0^s - 2^s) \vee 2^{s+1}) \bmod 2^d$, if $0 \leq s < d$ ($\vee$ is the inclusive-OR operator,)

Successor_1: $n_0^s + 2^{s-1}$, if $1 \leq s \leq d$,

Successor_2: $n_0^s - 2^{s-1}$, if $1 \leq s < d$.

Figure 2 illustrates two possible SBN communication patterns, but many others can easily be derived by slightly altering the SBN definition to fit a given network topology and application requirements. Multiple randomly-selected SBN patterns help distribute messages more evenly, enhance network reliability, and allow various applications to be written using different communication patterns. For example, the SBN communication pattern for processor 0 can be defined with the help of a one-dimensional array implementation of a full binary tree such that the predecessor and successors of a processor are given by:

Predecessor: $\left\lfloor \frac{n_0^s}{2} \right\rfloor$, if $0 \leq s < d$,

Successor_1: $2 \times n_0^s + 1$, if $1 \leq s \leq d$,

Successor_2: $2 \times n_0^s$, if $1 \leq s < d$.

The SBN definition is now modified to accomplish an embedding onto a hypercube topology, as will be required for the Hypercube Variant of the algorithm. This embedding uses a *modified binomial spanning tree*, which is actually two binomial trees[1] connected back to back. Figure 3 shows such a communication pattern for a 16-processor network which is used to route messages originating from processor 0. The solid lines of the diagram represent the actual SBN pattern, whereas the dashed lines are used to gather load balancing messages at a single destination processor (processor 15, in this case.) This embedding ensures that all successors and predecessors at any communication stage are adjacent processors in the hypercube. Also, every originating processor has a unique destination. Finally, if the processors are numbered using a binary string of d bits, the number of predecessors for a processor x is max $\{1, b\}$, where b is the number of consecutive leftmost 1-bits in the binary identifier of x.

[1] A binomial tree $B(k)$ is an ordered tree defined recursively as follows [13]: $B(0)$ consists of a single processor; $B(k)$ consists of two $B(k-1)$s linked together such that the root of one is the leftmost child of the root of the other.

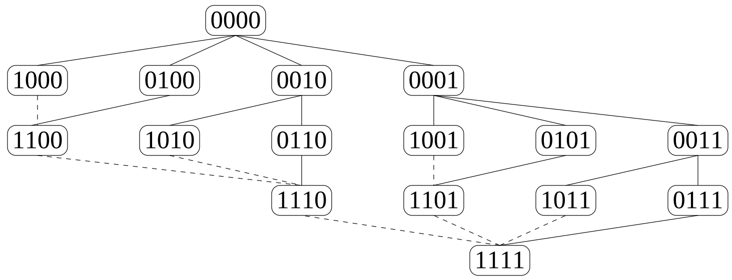


Figure 3. Modified binomial spanning tree used as a hypercube SBN.

3.2.3 SBN-based load balancers

Some general features that are shared by all three SBN-based dynamic load balancing algorithms are described in this chapter. Also described are various system thresholds, the two types of messages that are processed by the SBN approach, and pseudo codes for the procedures common to all three algorithms.

System thresholds

In general, many load balancing algorithms are very susceptible to the choice of system thresholds [63]. A proper selection of these threshold values has proven helpful in optimizing the SBN-based algorithms as well.

The proposed SBN-based algorithms adapt their behavior to the system load. Under heavy (respectively, light) loads, the balancing activity is primarily initiated by processors that are lightly (respectively, heavily) loaded. This activity is controlled by two system thresholds: `MinTh` and `MaxTh`, which are respectively the minimum and the maximum system load levels. The system load level, `SysLL`, is the average number of jobs queued per processor. If a processor p has a queue length $\texttt{QLen}(p) < \texttt{MinTh}$, load balancing is initiated. If, on the other hand, $\texttt{QLen}(p) > \texttt{MaxTh}$, the extra jobs $\texttt{ExLoad} =$

`QLen`$(p)-$ `MaxTh` are distributed through the network, without using explicit load balancing messages. However, if this distribution overloads a processor in the final stage (stage 0), a full load balancing operation is triggered.

The performance of the algorithms is affected by the chosen values for `MinTh` and `MaxTh`. For instance, `MinTh` must be large enough to receive sufficient jobs before a lightly-loaded processor becomes idle. However, it should not be too large to initiate unnecessary load balancing. Similarly, if `MaxTh` is too small, it will cause an excessive number of job distributions, and if it is too large, jobs will not be adequately distributed under light system loads. Basically, once there is sufficient load on the network, very little load balancing activity should be required.

Message types

Two types of messages are processed by the SBN-based algorithms. The first type is a *balancing message* which originates from an unbalanced processor and is then routed through the SBN. The cumulative total of queued jobs, `TotalJQ`, is computed to obtain a snapshot value of `SysLL`.

The second type of message is a *distribution message*, which is used during a load balancing operation to route the `TotalJQ` value though the network and to reassign jobs after the balancing message is broadcast. Each processor upon receipt of such a message, updates its local values of `SysLL`, `MinTh`, and `MaxTh`. Distribution messages can also be sent to predecessor processors prior to completing the broadcast of the balancing message. This action occurs so that jobs can be assigned to predecessor processors having less than `MinTh` jobs queued and about to go idle. Finally, distribution messages are sent when a load balancing operation is not in progress and a processor has greater than `MaxTh` jobs queued. In this case, `ExLoad` jobs are reassigned.

If the communication from one processor to its neighbors is completed in constant time, a single load balancing operation requires $O(\log P)$ time since there are $d + 1 = (\log P) + 1$ communication stages in SBN(d). However, if multiple balancing operations are processed simultaneously, the worst case communication complexity can be shown to be $O(\log^2 P)$ [16]. To reduce message traffic, a processor does not initiate additional load balancing activity until all previous balancing messages passing through it have been serviced.

Common procedures

All three of the load balancing algorithms consist of four key procedures. The first two, *GetBalance* and *GetDistribute*, are used to process balancing and distribution messages that are received, while the other two, *Balance* and *Distribute*, route those messages to the successors in the SBN. Implementation details of these procedures depend on the particular choice of load balancing algorithm. Figure 4 presents the pseudo code that is common to the SBN-based load balancing algorithms, and is executed in parallel by every processor.

The *UpdateLoad* procedure adjusts the system thresholds described in section 3.2.1. It is called by the procedures, *GetBalance* and *GetDistribute*, and completes the load balancing operation at each processor.

Note that `ConstParam` is a user-defined parameter that is used to set the minimum `MinTH` value. In the experiments performed, it was found that setting `ConstParam` = 2 proved to be the most effective value. However, the optimal setting of `ConstParam` could vary based on job characteristics. The `MinTh` setting minimizes the possibility that processors will go idle before the system load is balanced. The setting of `MaxTh` grows exponentially with `SysLL` because the need for load balancing activity decreases rapidly as `SysLL` increases. The mathematical justification for this policy is presented in section 3.3.4.

```
Procedure Main
Repeat forever
  Call GetBalance to process balancing messages received
  Call GetDistribute to process distribution messages received
  If (QLen(p) > MaxTh)
    ExLoad = QLen(p) − MaxTh
    Call Distribute to route ExLoad jobs through the SBN
  If (QLen(p) < MinTh)
    Call Balance to initiate a balance operation and determine TotalJQ
  Resume processing of application program
End Repeat

Procedure UpdateLoad(TotalJQ)
SysLL = ⌈ TotalJQ / P ⌉
MaxTh = SysLL + 2^⌊ SysLL / ConstParam ⌋
MinTh = SysLL − 1
If (SysLL > ConstParam) MinTh = ConstParam
Return
```

Figure 4. Common pseudo code for SBN-based load balancing algorithms.

3.3 Proposed algorithms

Based on the SBN concept, a baseline dynamic load balancing strategy and two variants are proposed in this dissertation. The Basic algorithm and the Hypercube Variant attempt to accurately compute and maintain the value of `SysLL`, whereas the Heuristic Variant estimates it.

3.3.1 Basic SBN algorithm

In this algorithm, balancing messages are routed through the SBN(d) from the source (root) at stage d to all the processors at stage 0. These messages are then routed back to the root so that `TotalJQ` can be computed. Thus, the originating root processor

```
Procedure GetBalance
While (balancing messages remain to be processed)
  If (QLen(processor sending this message) < MinTh)
       Route ⌊ QLen(p) / 2 ⌋ jobs to processor sending this message
  If (no balance operation active from the SBN root processor)
       Increment count of simultaneous balance operations being serviced
       If (Stage(p) == 0) Call Balance to route QLen(p) value towards root processor
       Else  Indicate two balancing messages to be gathered from successors
             Call Balance to route balancing message to successors
  Else
       If (second balancing message is to be gathered from successors) Continue
       If (Stage(p) == d)
             Call UpdateLoad(TotalJQ)
             ExLoad = QLen(p) − SysLL
             Call Distribute to route ExLoad jobs & TotalJQ value to successors
             Decrement count of simultaneous balance operations being serviced
       Else Call Balance; route (QLen(p) + QLen(successors)) value towards root processor
End While
Return

Procedure GetDistribute
While (distribution messages remain to be processed)
  QLen(p) = QLen(p) + JobsRecv
  If (TotalJQ value contained in message received)
       Call UpdateLoad(TotalJQ)
       Route min(QLen(p) − SysLL, SysLL − QLen(predecessor)) jobs to predecessor
       Decrement count of simultaneous balance operations being serviced
  If (Stage(p) == 0)
       If (QLen(p) > MaxTh) Call Balance to initiate new balance operation
  Else ExLoad = QLen(p) − MaxTh
       If (this is a balance operation) ExLoad = QLen(p) − SysLL
       Call Distribute to route ExLoad jobs and/or TotalJQ value to successors
End While
Return

Procedure Balance
If (Stage(p) == d)
  If (balance operation already underway) Return
  Increment count of simultaneous balance operations being serviced
  Indicate that a balancing message is expected from successor
Route the balancing message to next SBN stage
Return

Procedure Distribute
If ((this is a non-balance distribution) And (balance operation is underway))
  Inhibit the distribution and Return
QLen(p) = QLen(p) − ExLoad
If ((ExLoad > 0) Or (TotalJQ value to send)) Route distribution message to successors
Return
```

Figure 5. Basic SBN algorithm pseudo code.

has an accurate snapshot value of SysLL. Next, distribution messages are sent to relocate jobs and to broadcast the TotalJQ value. All processors then update their local SysLL, MinTh, and MaxTh. The extra load (ExLoad) of jobs are routed as part of this distribution to balance the system load. In addition, if QLen$(p) <$ SysLL for a processor p, the need for jobs is indicated during the distribution process. Successor processors respond by routing back an appropriate number of excess jobs, if available. Figure 5 presents the pseudo code of the Basic SBN algorithm where Stage(p) denotes the SBN stage of processor p for a given communication pattern (see figure 2) and JobsRecv is the number of jobs received by it in a distribution message.

To illustrate the operation of this algorithm, consider SBN(3) in figure 6(a). The label and QLen(p) for each processor p are respectively shown inside and outside the corresponding circle. For example, processor 6 has $Q = 3$ jobs queued for execution. At processor 0, initial values are SysLL $= L = 4$, MinTh $= m = 2$, and MaxTh $= M = 6$. After a load balancing request is sent through the SBN and then routed back to processor 0, these values are updated as $L = 8$, $m = 2$, and $M = 24$, using the *UpdateLoad* procedure given in figure 4. Note that when the balancing is initiated, processor 4 distributes half of its QLen(p) jobs (i.e., $\lfloor 3/2 \rfloor = 1$) back to processor 0. This distribution is shown by the label on the arrow in figure 6(a).

Distribution messages are then used to route excess jobs to the successor processors or to indicate a need for jobs if the QLen$(p) <$ SysLL. Jobs are routed back to the predecessors when appropriate. Figure 6(b) shows the result of this distribution where the labels on arrows indicate the number of jobs routed between processors.

To balance P processors, $P - 1$ balancing messages are first sent through the SBN, which are then routed back to the root processor so that the SysLL value can be computed through a global reduction operation. Finally, $P - 1$ distribution messages are sent to balance the load as well as broadcast the TotalJQ value through the network.

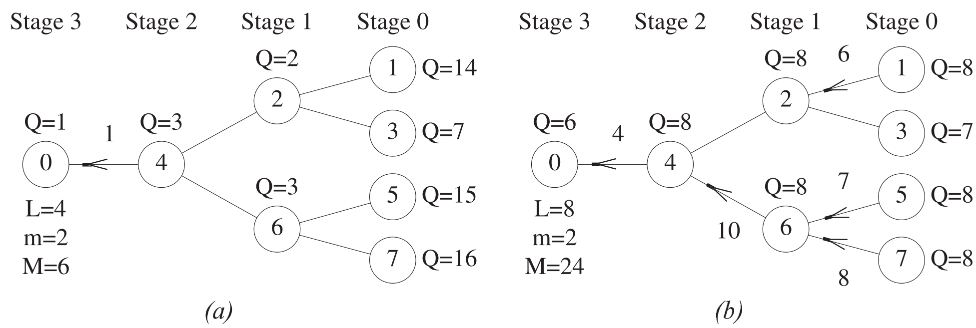


Figure 6. Load balancing example using Basic SBN algorithm. Jobs distributed; (a) during balancing, (b) during distribution.

Note that, if a processor has an immediate need for jobs that can be supplied during load balancing, additional distribution messages are sent from neighboring processors to satisfy the need. If J such messages are required, a total of $3P - 3 + J$ messages will be have to be processed. Since $J = O(P)$, the total number of messages to be processed is $O(P)$. The depth of the SBN network being $O(\log P)$, the total time required to complete a load balance operation is $O(\log P)$.

3.3.2 Hypercube variant

The SBN approach was adapted for implementation on a hypercube topology, using the modified binomial spanning tree illustrated in figure 3. This algorithm uses the same control variables (`SysLL`, `MinTh`, and `MaxTh`) and processes balancing and distribution messages in the same manner as the Basic SBN algorithm. However, in embedding the SBN onto a hypercube, (see figure 3,) the processor r at stage d sends a balance message to each of its adjacent processors at stage $d-1$. These messages are then routed through the hypercube network and eventually collected at the single destination processor q at stage 0. Processor q then accurately computes the current `SysLL` value and initiates the distribution process by routing distribution messages back towards the root processor at stage d. Note that the exclusive-OR property described in section 3.2.2 still holds. This property allows any processor to correctly determine the successors and predecessors if the stage number and the root processor are given.

Other differences between the Hypercube Variant and the Basic SBN algorithm are as follows.

- In the Hypercube Variant, the value of `TotalJQ` is computed when all balancing messages arrive at the destination processor. Unlike the Basic algorithm, this is possible because there is a unique destination for every originating processor in the hypercube embedding. Distribution messages are then routed back to complete the load balancing. Since there are $P - 1 + \frac{P}{2} - 1$ interconnections in the modified binomial spanning tree (see figure 3,) a load balancing operation requires $3P - 4$ messages excluding the distribution messages sent between neighbors to satisfy the immediate need for jobs.

- Balancing messages always proceed from the root processor (at SBN stage d of the Hypercube Variant) towards stage 0. This contrasts with the Basic SBN algorithm where balancing messages first proceed from stage d to stage 0, and are then routed back to stage d.

- To minimize the communication overhead in the Hypercube Variant, messages are gathered from the previous stage whenever more than one message is expected. The Basic SBN algorithm only needs to gather messages that are being routed back towards to root processor (because messages going in the other direction have only one predecessor.)

- The network topology for the Hypercube Variant is such that the number of predecessor and successor processors vary at different communication stages.

Consider the hypercube of dimension three shown in figure 7 (a) and $7(b)$ where the symmetric broadcast communication paths are shown originating from node 3 as the root to the other nodes of the network. The label and $\texttt{QLen}(p)$ for each processor p are respectively shown inside and outside the corresponding circle. For example, processor 6 has $Q = 10$ jobs queued for execution. At processor 3, initial values are $\texttt{SysLL} = L = 6$, $\texttt{MinTh} = m = 2$, and $\texttt{MaxTh} = M = 14$. After a load balancing request originating from processor 3 is sent through the SBN, the *UpdateLoad* procedure in figure 4 is used to updatethese values as $L = 7$, $m = 2$, and $M = 15$.

Note that as the balancing message is sent through the SBN, processor 3 embeds a need for two jobs from each of its successors. The following formula is used for this calculation (which is different that that used by the Basic SBN algorithm:)

$$\texttt{JobRQ} = \lceil (\texttt{ExLoad})/(\mathrm{n}umber of successors) \rceil$$

Only processor 7 can immediately return jobs upon receipt of the balance message since it is the only successor to processor 3 with $\texttt{QLen}(p) > \texttt{SysLL}$. This distribution is shown by the label on the arrow in figure $7(a)$.

Distribution messages are then used to route excess jobs to the successor processors or to indicate a need for jobs if the $\texttt{QLen}(p) < \texttt{SysLL}$. Jobs are routed back to the predecessors when appropriate. Figure $7(b)$ shows the result of this distribution where the labels on arrows indicate the number of jobs routed between processors.

3.3.3 Heuristic variant

Both the Basic algorithm and the Hypercube Variant are expensive since a large number of messages has to be processed to accurately maintain the $\texttt{SysLL}$ value. The Heuristic Variant attempts to reduce this overhead by terminating load balancing operations as soon as $\texttt{QLen}(p)$ is sufficiently large. In general, this strategy reduces the number of messages although $O(P)$ messages are still needed in the worst case. The pseudo code

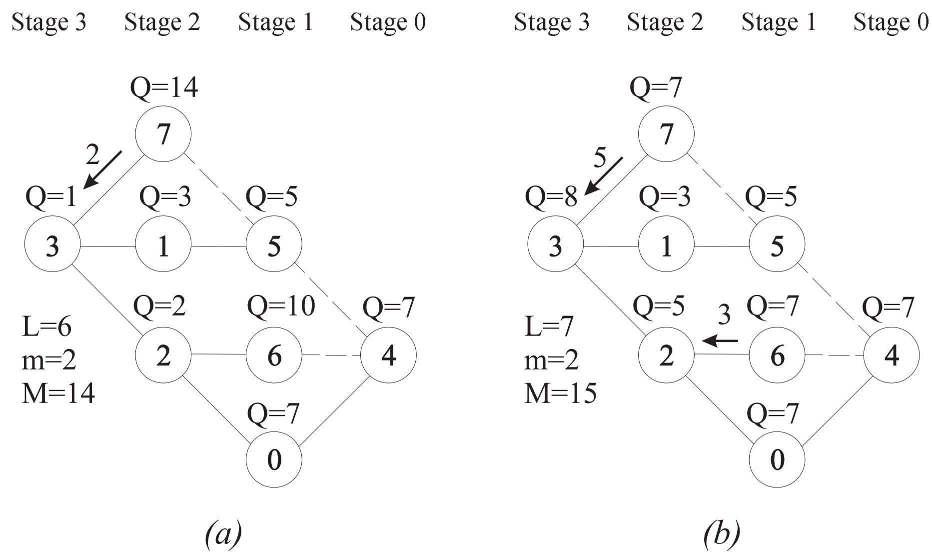


Figure 7. Load balancing example using Hypercube Variant. Jobs distributed; (a) during balancing, (b) during distribution.

in figure 8 gives the operational details of the Heuristic Variant of the SBN-based load balancing algorithm.

Similar to the Basic algorithm, load balancing is initiated by processor p when $\texttt{QLen}(p) < \texttt{MinTh}$ by sending a balancing message to its SBN successor. However, in the Heuristic Variant, the processor that receives this message estimates $\texttt{SysLL}$ by averaging $\texttt{QLen}(p)$ over the processors through which the balancing message has already passed. If $\texttt{QLen}(p) > \texttt{SysLL}$, an appropriate number of jobs ($\texttt{ExLoad} = \texttt{QLen}(p)/2$) is then returned via a distribution message as shown in the *GetBalance* procedure in figure 8. If $\texttt{QLen}(p) \leq \texttt{SysLL}$, the balancing message is forwarded to the next SBN stage. Otherwise, the load balancing procedure is terminated. Detailed mathematical justifications for the Heuristic Variant are discussed later in section 3.3.4.

```
Procedure GetBalance
While (balancing messages remain to be processed)
  TotalJQ = ⌈ P × (QLen(p) + QLen(predecessors)) / (d − Stage(p) + 1) ⌉
  OldSysLL = SysLL
  Call UpdateLoad(TotalJQ)
  ExLoad = ⌊ QLen(p) / 2 ⌋
  If (ExLoad > 0)
    If (QLen(p) ≤ SysLL)
        Call Balance to route balancing message to successors
      Else
        Call Distribute to route ExLoad jobs to the predecessor
End While
Return

Procedure GetDistribute
While (distribution messages remain to be processed)
  JobsQueued = QLen(p)
  QLen(p) = QLen(p) + JobsRecv
  If (distribution towards the root processor)
    TotalJQ = P × (JobsQueued + ⌈ JobsRecv / (d − Stage(p) + 1) ⌉ )
    Exload = ⌊ QLen(p) / 2 ⌋
  Else TotalJQ = P × (JobsQueued + ⌈ JobsRecv / (2^(Stage(p)+1) − 1) ⌉ )
    ExLoad = QLen(p) − SysLL
  Call UpdateLoad(TotalJQ)
  If (ExLoad > 0)
    Call Distribute to route ExLoad jobs to next SBN stage
End While
Return

Procedure Balance
If (Stage(p) == 0) Return
If (Stage(p) == d)
  TotalJQ = P × QLen(p)
  Call UpdateLoad(TotalJQ)
Route the balancing message to successors
Return

Procedure Distribute
If (Stage(p) == 0) Return
If (Stage(p) == d)
  TotalJQ = P × (SysLL + ⌈ QLen(p) − SysLL / P ⌉ )
  Call UpdateLoad(TotalJQ)
  ExLoad = QLen(p) − SysLL
QLen(p) = QLen(p) − ExLoad
Route ExLoad jobs with the distribution message to next SBN stage
Return
```

Figure 8. Heuristic Variant pseudo code.

Job distribution is also accomplished differently in this Heuristic Variant. Suppose $\texttt{QLen}(p) > \texttt{MaxTh}$, which implies a job distribution is necessary. A new value of $\texttt{SysLL}$ is estimated as:

$$\texttt{SysLL} = \texttt{SysLL} + \lceil (\texttt{QLen}(p) - \texttt{SysLL}) \ / \ P \ \rceil.$$

Intuitively, processor p tries to distribute the excess load among all the processors. Each processor r receiving the distribution message adjusts its $\texttt{SysLL}$ value and distributes excess jobs. The jobs that are received ($\texttt{JobsRecv}$) are distributed evenly among r and all others in the remaining SBN stages. This is reflected in the formulae shown in the *GetDistribute* procedure in figure 8. Note that $\texttt{SysLL}$ is updated based on $\texttt{QLen}(r)$ and not the original value of $\texttt{SysLL}$. This approach proved to be more effective in the experiments that were performed.

For example, consider an SBN(3) that has a processor p with $\texttt{SysLL}= 7$, $\texttt{MaxTh}= 15$, and $\texttt{QLen}(p) = 24$. The newly-computed $\texttt{SysLL}$ value is $7 + \lceil (24 - 7)/8 \rceil = 10$. The number of jobs that will be distributed to the successor (only one successor at this stage) is $24 - 10 = 14$. Suppose that the successor r has $\texttt{SysLL}= 9$, and $\texttt{QLen}(q) = 6$. After receiving 14 jobs, it has $\texttt{SysLL}= 6 + \lceil 14/7 \rceil = 8$. Thus, $20 - 8 = 12$ excess jobs will be distributed to the next stage.

A significant advantage of the Heuristic Variant is that the balancing messages do not have to be gathered in order to calculate $\texttt{SysLL}$. This reduces the interdependencies associated with the communication and allows fault tolerance. If a particular processor fails, load balancing can still be accomplished with the help of the remaining processors.

An additional improvement for the Basic and Heuristic Variant of the load balancing algorithms can be obtained by using multiple communication patterns in the SBN. Each time a message is initiated, one of the SBN patterns is randomly chosen. The experiments make use of the two communication patterns mentioned in section 3.2.2 for computing predecessors and successors. Each of the balancing and distribution messages includes the source processor, the pattern used, and the stage to which the message is being routed. Since all processors have the SBN template associated with messages originating from processor 0, the required SBN communication pattern can be determined.

3.3.4 SBN message passing analysis

In a multicomputer of P processors, the distribution of jobs among processors can be modeled using a Poisson distribution. Specifically, the probability of a processor having j jobs is given by $\frac{\lambda^j}{e^{\lambda} j!}$, where λ is the mean arrival rate. If `SysLL` $= k$, then by definition, the average number of jobs assigned to a processor is k. Hence, the probability that a processor has j jobs is $\frac{k^j}{e^k j!}$. Using this simple model, useful probabilities can be easily calculated. For example, the probability, g_3, that a processor in the network has more than three jobs, is $g_3 = 1 - \frac{1}{e^k}(1 + k + \frac{k^2}{2} + \frac{k^3}{6})$. The probability that all P processors have more than three jobs is $(g_3)^P$. Now, $(g_3)^P > 0.9$ if $k > 5$, and is almost unity if $k > 15$. This implies that the need for load balancing activity rapidly decreases as `SysLL` increases. Therefore, it makes sense to increase `MaxTh` exponentially as `SysLL` increases.

To analyze the Heuristic Variant, let us compute the expected number of jobs, `EJobs`, that are returned to the processor that initiates the SBN load balancing algorithm. Let us also compute the expected number of processors, `EProcs`, that will be visited during a balancing operation. In this way, it can be determined whether sufficient jobs are returned and if message traffic is reduced by utilizing the Heuristic Variant. In this section, SBN message passing is modeled mathematically. For this purpose, let us define the following four probability vectors:

- $\Phi = < \phi_0, \phi_1, \cdots, \phi_n >$, where ϕ_i is the probability that a processor has i jobs queued for a given value of `SysLL`.

- $\Phi_{stop} =< 0, 0, \cdots, 0, \phi_{stop}, \phi_{stop+1}, \cdots, \phi_n >$ defines the likelihood that balancing will stop at a given processor. Here, *stop* is the number of jobs queued at a processor that will prevent the SBN Heuristic Variant from forwarding a balancing message to the next stage.

- $\Phi_{continue} =< \phi_0, \phi_1, \cdots, \phi_{stop-1}, 0, \cdots, 0 >$ defines the likelihood that balancing messages will be sent to successor processors.

- $C(V)$ is a probability vector computed by applying the function C to the probability vector V, and indicates the number of jobs distributed to a processor's predecessor. Here, V is the vector defining the probabilities of jobs queued at a processor after receiving distribution messages from its successors.

Proposition 1 *If $V_1 =< v_{10}, v_{11}, \cdots, v_{1n} >$ and $V_2 =< v_{20}, v_{21}, \cdots, v_{2m} >$ are probability vectors, then $V_1 \bigotimes V_2 =< v_0, v_1, \cdots, v_{m+n} >$, where $\bigotimes$ is a product operation such that $v_i = \sum_{j+k=i} v_{1j} \times v_{2k}$, is also a probability vector.*

Proof. Consider the product $(\sum_{i=0}^{n} v_{1i})(\sum_{j=0}^{m} v_{2j})$. Since V_1 and V_2 are probability vectors, $\sum_{i=0}^{n} v_{1i} = \sum_{j=0}^{m} v_{2j} = 1$ and the above product is unity. This product can also be written as $v_{10}(v_{20} + v_{21} + \cdots + v_{2m}) + \cdots + v_{1n}(v_{20} + v_{21} + \cdots + v_{2m})$. These terms can be reorganized as $(v_{10}v_{20}) + (v_{10}v_{21} + v_{11}v_{20}) + (v_{10}v_{22} + v_{11}v_{21} + v_{12}v_{20}) + \cdots,)$ which are the vector entries of $V_1 \bigotimes V_2$. Since it was shown that this sum is unity, $V_1 \bigotimes V_2$ is also a probability vector. $\square$

As an example, let $V_1 =< 0.5, 0.3, 0.2 >$ and $V_2 =< 0.4, 0.3, 0.3 >$. Then,

$$V_1 \bigotimes V_2 =< 0.20, 0.15 + 0.12, 0.15 + 0.09 + 0.08, 0.09 + 0.06, 0.06 >=< 0.20, 0.27, 0.32, 0.15, 0.06 > .$$

Clearly, the sum of all the vector entries of $V_1 \bigotimes V_2$ is equal to unity.

Using the definitions just described, define $R_k =< j_{k0}, j_{k1}, \cdots, j_{kn} >$ as a probability vector, where j_{ki} is the probability that i jobs from an SBN processor at stage k

are returned to a predecessor at SBN stage $k+1$ during a balancing operation. R_{d-1} then indicates the jobs returned to the root processor that initiated the load balancing operation, and can be computed by the following recursive formula:

Stage 0: $R_0 = C(\Phi)$. Note that the function C reflects the SBN job return policy as charted in figures 9 and 10.

$$Stage\ 1 \leq k < d : R_k = C(\Phi_{stop} + \Phi_{continue} \bigotimes R_{k-1} \bigotimes R_{k-1}).$$

Intuitively, the calculation function is used to add the jobs queued locally to the jobs that are expected to be returned from the two successor processors. Once $R_{d-1} =< r_0, r_1, \cdots r_n >$ is computed, `EJobs` $= \sum_{j=0}^{n} j \times r_j$ can be calculated.

To compute `EProcs`, let ϕ_c be the probability that a given processor has greater than or equal to *stop* number of jobs queued, where $c = \phi_{stop} + \phi_{stop+1} + \cdots + \phi_n$ in the Φ_{stop} probability vector. The probability that a balancing message will reach a processor at stage k in the SBN is therefore p_c^{d-k-1}, since the message must pass through $d-k-1$ predecessors. By definition, for a given SBN communication pattern, the number of processors at stage k is 2^{d-k-1}. Therefore, `EProcs` $= \sum_{k=0}^{d-1} \phi_c^{d-k-1} \times 2^{d-k-1}$. For example, if $d = 5$ and $\phi_c = 0.4$, the expected number of the thirty two processors visited during a balancing operation is 3.3616.

With this model established, `EJobs` and `EProcs` can be computed for various values of the system load level (`SysLL`), the dimension d of the SBN, and the value of *stop*. Assuming a Poisson job arrival rate, the probability vector Φ is easily established. The calculation function, $C(V)$, is implemented by utilizing the job return policy of the Heuristic Variant.

Figures 9 and 10 utilize the mathematical model just described to depict `EJobs` and `EProcs`, when $P = 32$, for various `SysLL` values. In these tables and graphs, several prospective policies are analyzed for a processor to terminate load balancing. For example, when a processor p executes the Basic SBN algorithm after receiving a balancing message, this message will be forwarded to the next SBN stage unless p is at stage 0. An alternate policy is for p to terminate balancing if $\texttt{QLen}(p) \geq 2$ when SysLL is low (say, ≤ 2) or when $\texttt{QLen}(p) \geq 4$ for other SysLL values. This policy attempts to return at least one job under light loads and at least two jobs under heavier loads. Other possible policies are for p to stop balancing when $\texttt{QLen}(p)$ is sufficiently above SysLL. These policies are depicted up to $\texttt{QLen}(p) = \texttt{SysLL}+4$. In the figures, SysLL is varied from a value of one to fifteen.

Figure 10 shows that if the Heuristic Variant is used with $stop = \texttt{SysLL}+1$, EProcs is significantly less than with the Basic SBN algorithm. For example, if $P = 32$ and SysLL $= 4$, only 8.3 processors are visited, on average, using the Heuristic Variant, whereas all 31 processors are required for the Basic algorithm. Furthermore, when the Heuristic Variant is used, figure 9 indicates that EJobs= 4.2 (compared to 8.8 with the Basic algorithm), a value much closer to the SysLL value of 4. Therefore, a better load balance is achieved with significantly reduced balancing traffic.

Figure 9 also gives an indication as to what *stop* value is optimal in relation to SysLL. After a successful load balancing operation, $\texttt{QLen}(p)$ will approximate the SysLL value for all processors. By the definition of SBN load balancing algorithms, the processor that initiates load balancing has less than MinTh jobs queued for processing. EJobs will then be optimal when it approximates the SysLL value. Figure 9 shows that this objective is achieved when $stop = \texttt{SysLL}+1$. This is intuitively correct because it implies that a processor will forward a load balancing message to the next SBN stage until an overloaded processor is encountered.

SysLL Value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Basic SBN Balancing Algorithm	1.3	3.8	6.3	8.8	11.3	13.8	16.3	18.8	21.3	23.8	26.3	28.8	31.3	33.8	36.3
Stop if QLen(P) >= SysLL + 4	1.3	3.6	5.8	7.9	9.8	11.7	13.3	15.0	16.6	18.2	19.6	21.1	22.5	24.0	25.3
Stop if QLen(P) >= SysLL + 3	1.3	3.4	5.3	7.0	8.7	10.1	11.6	13.0	14.4	15.7	17.0	18.3	19.6	20.8	22.1
Stop if QLen(P) >= SysLL + 2	1.1	2.9	4.3	5.8	7.0	8.3	9.5	10.8	11.9	13.1	14.2	15.4	16.5	17.7	18.8
Stop if QLen(P) >= SysLL + 1	0.8	1.9	3.2	4.2	5.4	6.4	7.5	8.5	9.6	10.6	11.7	12.7	13.8	14.7	15.8
Stop if QLen(P)>=2 if SysLL<=2, QLen(P)>=4 if SysLL>2	0.8	1.3	3.2	3.1	3.1	3.3	3.5	3.9	4.3	4.8	5.3	5.8	6.3	6.8	7.3

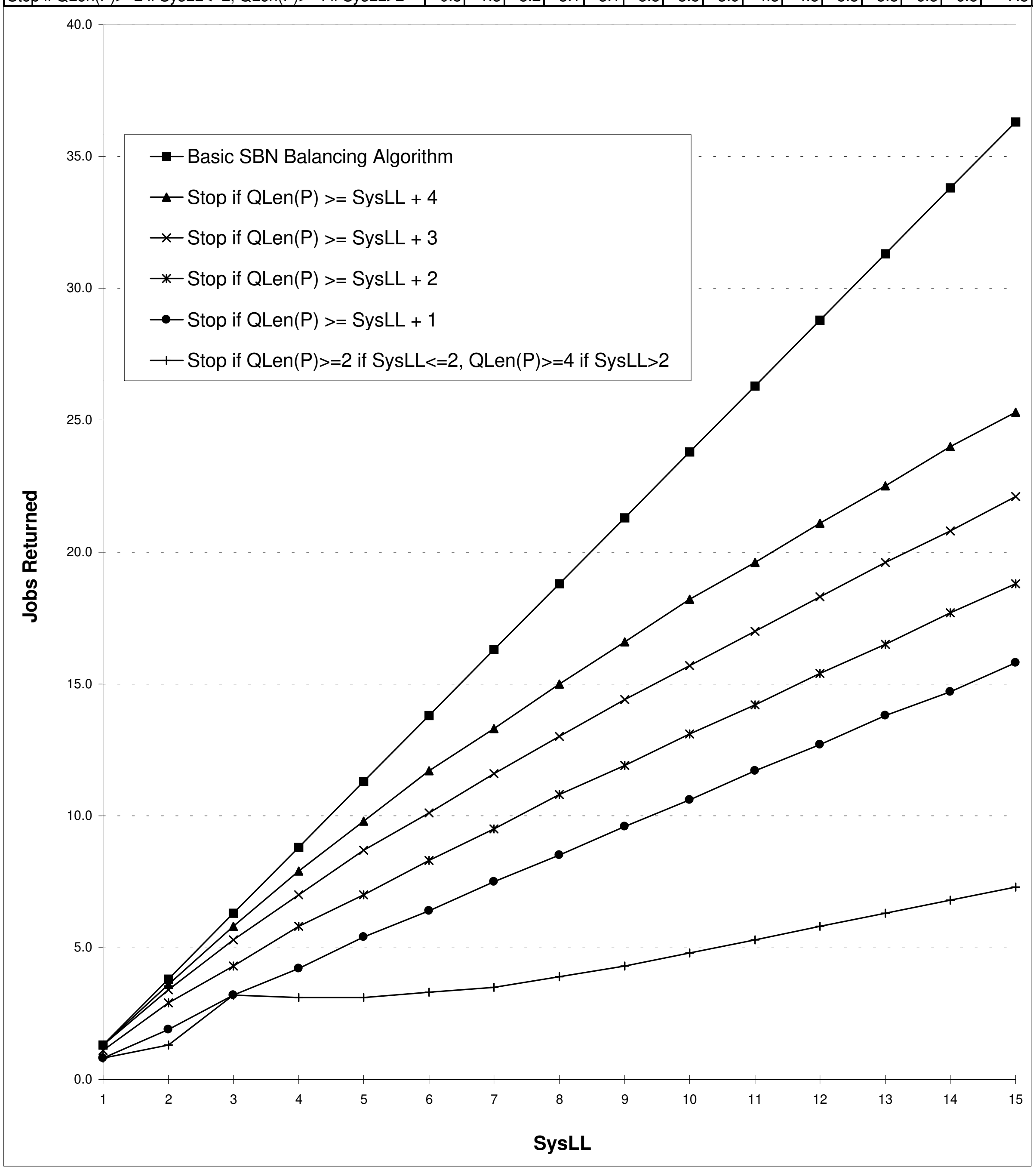


Figure 9. Expected number of jobs returned (EJobs) for $P = 32$.

SysLL Value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Basic SBN Balancing Algorithm	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0	31.0
Stop if QLen(P) >= SysLL + 4	30.6	29.4	27.9	26.3	24.9	23.6	22.5	21.5	20.6	19.8	19.1	18.5	17.9	17.4	16.9
Stop if QLen(P) >= SysLL + 3	29.2	26.2	23.6	21.6	20.0	18.7	17.6	16.7	16.0	15.3	14.8	14.3	13.9	13.5	13.1
Stop if QLen(P) >= SysLL + 2	23.9	19.3	16.7	15.0	13.8	12.9	12.2	11.7	11.2	10.8	10.5	10.2	10.0	9.7	9.5
Stop if QLen(P) >= SysLL + 1	12.5	10.0	8.9	8.3	7.9	7.6	7.4	7.2	7.1	7.0	6.9	6.8	6.7	6.6	6.6
Stop if QLen(P)>=2 if SysLL<=2, QLen(P)>=4 if SysLL>2	12.5	3.4	8.9	3.8	2.0	1.4	1.2	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0

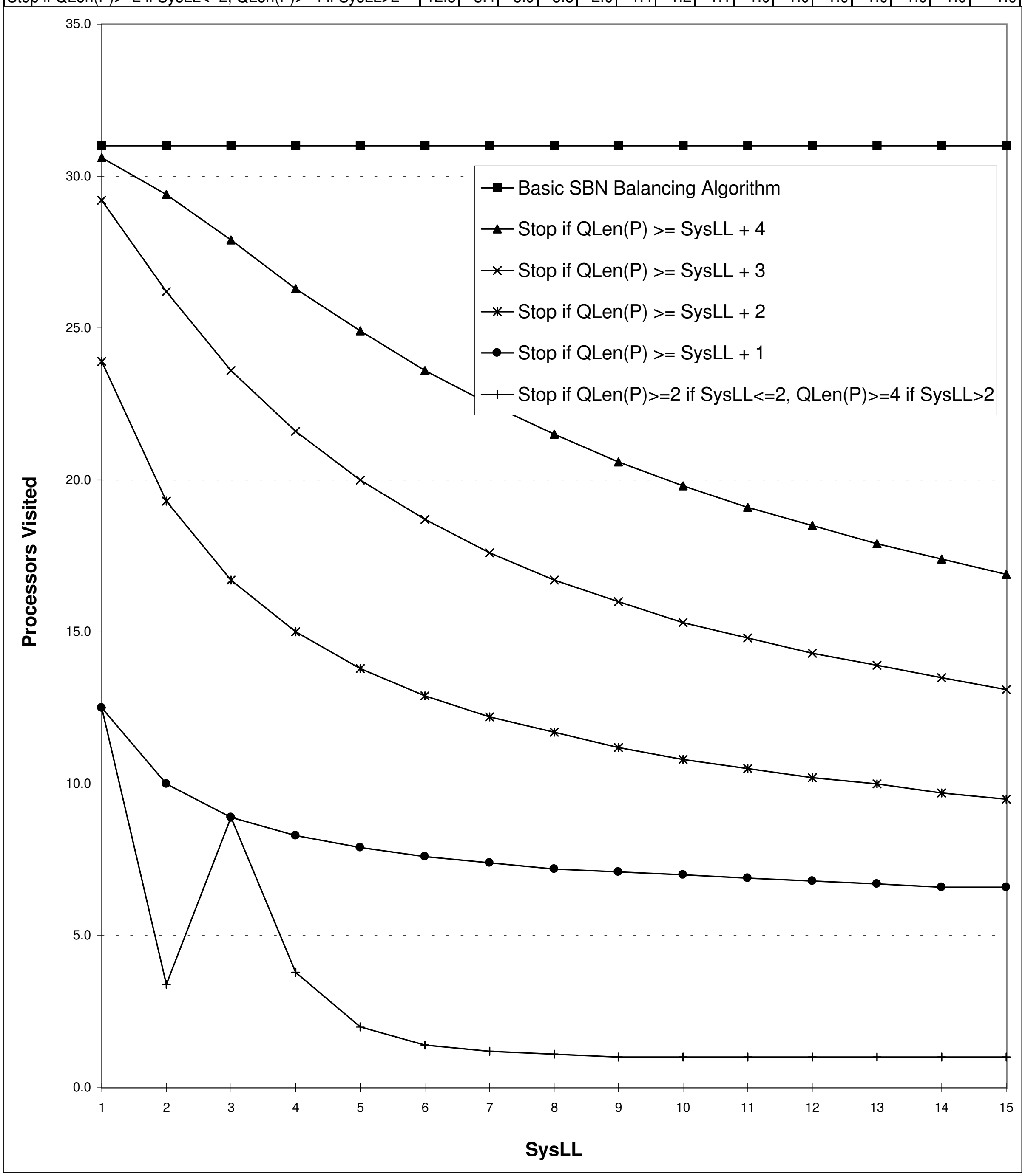


Figure 10. Expected number of processors visited (EProcs) for $P = 32$.

3.4 Experimental study

This section describes the simulation environment and the synthetic test cases that were used to compare the proposed SBN-based load balancing algorithms with several existing methods. The performance metrics used for comparison are explained and comprehensive results presented from simulation experiments.

3.4.1 Simulation environment

All the load balancing algorithms were implemented using MPI and tested with synthetically-generated workloads on a 32-processor SGI Origin2000 located at NASA Ames Research Center. The simulation program spawns the appropriate number of child processes and creates the desired SBN topology. A list of all processor labels and an initial distribution of jobs are then routed through the network.

In addition to the initial load, each processor dynamically generates additional jobs to be processed. Specifically, ten job creation cycles are run where the number of jobs generated at each processor during a cycle follows a Poisson distribution. By randomly picking different values of λ (the mean arrival rate,) varying numbers of jobs are created. Therefore, both heavy and light system load conditions are dynamically simulated. Jobs are processed by "spinning" for the designated amount of time. The simulation terminates when all jobs are processed. Note that as the number of processors increases, the workload also increases by the same factor. It is thus expected that the completion time should remain relatively constant if the algorithm scales efficiently.

The experiments compare the performance of the SBN-based load balancing algorithms to six other commonly-used techniques, namely Random, Gradient, Receiver Initiated, Sender Initiated, Adaptive Contracting, and Tree Walking, as summarized in section 3.1. The Basic SBN algorithm, its Heuristic Variant, and the Tree Walking algorithm utilize the SBN topology in their implementation. The SBN topology is very close to the binary tree structure required by Tree Walking and provides a direct and fair comparison. The other algorithms and the Hypercube Variant of the SBN algorithm are implemented utilizing a hypercube topology. This demonstrates the ability to embed the SBN approach into another topology and provides a fair comparison between the Hypercube Variant and the other balancing algorithms. By comparing the Hypercube Variant and Basic SBN algorithm, it can be determined whether a change in topology causes any significant performance differences. Finally, the same experiments are also performed without any load balancing, in order to have a reference point. All ten algorithms (six existing load balancing schemes, three proposed schemes, and a no-balancing scheme) are implemented using the same hardware and software environment.

The results obtained from these experiments are shown in figures 11, 12, and 13. They were compiled from repeating the simulations ten times and averaging the results. Experiments were limited to ten runs by system account limitations. The loads are characterized in the descriptions that follow:

Heavy System Load (see figure 11)

An initial load of ten jobs per processor is queued during the first cycle of execution. Each execution cycle, including the first, is 1.0 sec in duration. At the start of the remaining nine cycles, an average of 19.41 additional jobs are generated in accordance with the formula $200(\frac{\lambda^j}{e^\lambda \times j!})$, where the values of λ and j are randomly varied between one and ten. The constant, two hundred, is large enough to ensure that the time required to process the created jobs is approximately double the 10.0 secs. of execution (1.0 sec for each execution cycle,) thereby guaranteeing an overloaded condition. The duration of all jobs average 0.1 sec, with the longest jobs requiring 0.2 sec. The entire simulation requires 10.0 secs plus the time needed to empty out all of the job queues. Optimal balancing for this experiment requires an average of 1.0 sec to process the initial load plus 1.941×9 secs for processing the additional load cycles, amounting to 18.469 secs. The entire simulation is repeated ten times, and the results are averaged.

Heavy to Light System Load (see figure 12)

An initial load of fifty jobs per processor is queued to a small subset ($\log P$) of processors during the first execution cycle. This ensures the initial heavy load condition. Each execution cycle, including the first, is 4.0 secs in duration. At the start of the remaining nine cycles, an average of 12.94 additional jobs are generated in accordance with the formula $260(\frac{\lambda^j}{e^\lambda \times j!})$, where the values of λ and j are randomly varied between one and twenty. The constant, 260, is chosen so that a light load of jobs will be generated at each execution cycle. The duration of all jobs average 0.2 sec, with the longest jobs requiring 0.4 sec. The entire simulation requires 40.0 secs (4.0 secs for each execution cycle,) if load balancing is effective. Note that 10.0 secs is required to process the initial load plus $12.96 \times 9 \times 0.2$ secs for processing the remaining cycles. This totals to 33.328 secs, leaving an average of 0.66 sec of idle time per cycle. As with the other experiments, this entire experiment is repeated ten times, and the results are averaged.

Light System Load (see figure 13)

This experiment is similar to the Heavy to Light experiment except that the initial load of jobs is very light. Specifically, an initial load of one job per processor is queued to a small subset ($\log P$) of processors during the first cycle of execution. Therefore, a light system load exists throughout the duration of this experiment. The entire simulation requires 40.0 secs (4.0 secs for each execution cycle,) if load balancing is effective. Note that 0.2 sec is required to process the initial load plus $12.96 \times 9 \times 0.2$ secs to process the remaining cycles. This totals to 23.528 secs, leaving an average of 1.6472 secs of idle time per cycle. As with the other simulations, this entire experiment is repeated ten times, and the results are averaged.

3.4.2 Performance metrics

The data and bar charts included in figures 11, 12, 13 measure the comparative performance of the various load balancing algorithms on an SGI Origin2000. The X-axis of the bar charts show the number of processors used. The Y-axis tracks the following variables:

Message Traffic Comparison:

Total number of balancing and distribution messages that were sent during the simulation.

Total Jobs Transferred:

Total number of jobs that were transferred from one processor to another. Note that if a job is transferred multiple times before execution, each transfer is individually counted. However, if data needs to be transferred when a job executes on a processor other than the one at which it originated, the data are transferred only once. Note that it may have been appropriate to count multiple job transfers only once since the actual data transfer incurs almost all of the overhead. However, the Total Jobs Transferred metric as defined can be useful in that it can give an indication of how flexible an algorithm is in its ability to adapt to a quickly changing dynamic load environment. For example, load balancing algorithms that do not allow multiple transfers would be the least flexible and expected to generate the smallest Total Jobs Transferred values. Similarly, high Total Jobs Transferred values generated by a load balancing algorithm are not necessarily undesirable.

Maximum Variance in Idle Time:

Difference in processing time (in secs) between the busiest processor and the least busy processor.

Total Time to Complete:

Total amount of elapsed time (in secs) before all jobs are fully processed.

3.4.3 Summary of results

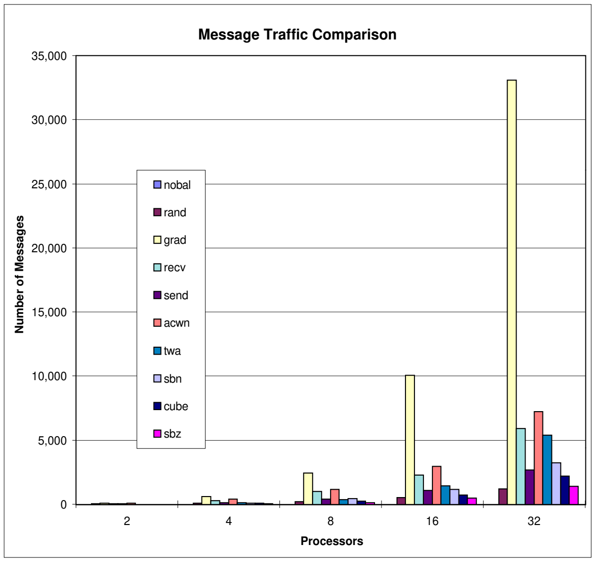
As mentioned in section 3.4.1, the Basic SBN-based load balancing algorithm (in short, SBN), its Heuristic Variant (SBZ), and the Tree Walking algorithm (TWA) were implemented using the SBN topology, while the other load balancing algorithms were implemented assuming a hypercube topology. Recall that the Hypercube Variant of SBN (CUBE) utilizes the Basic SBN algorithm adapted for the hypercube. Analyzing CUBE, it can be determined whether performance improvements are due to the proposed load balancing algorithms or due to the SBN topology.

The following paragraphs analyze each of the four metrics (Maximum Variance in Idle Time, Total Time to Complete, Message Traffic Comparison, and Total Jobs Transferred) measured during the experiments.

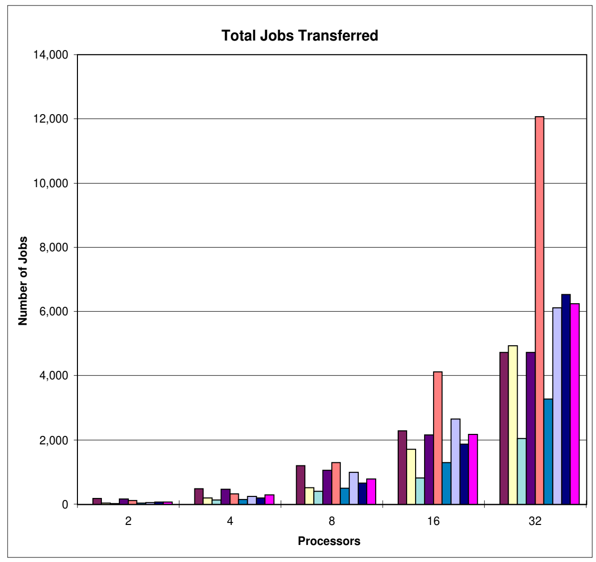
With respect to the metric Maximum Variance in Idle Time, the no-balancing algorithm (NOBAL) performs the worst by far, as expected. The Random (RAND) algorithm, although reducing the idle time, is much less effective than the other algorithms. The Sender Initiated (SEND) and Adaptive Contracting (ACWN) algorithms have similar performance, which is somewhat better than RAND. Note that RAND, ACWN, and SEND do not allow multiple job migrations. This feature prevents these algorithms from efficiently adapting to a dynamically-changing job execution environment. TWA shows a large imbalance under light system loads (see figure 13) that could stem from its lack of using a minimum processor workload threshold.

Considering the Total Time to Complete as the metric, recall that the amount of work increases proportionately with the number of processors. For example, twice as much processing is required for $P = 8$ than for $P = 4$. Under light loads (see

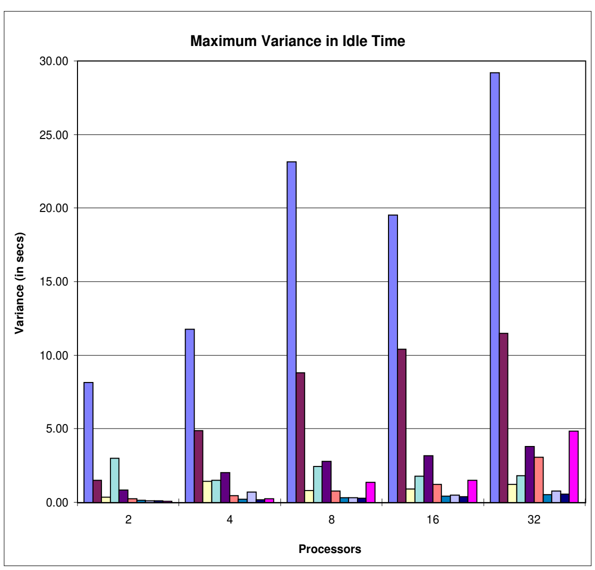
P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	20	63	25	34	99	18	12	14	8
4	0	71	586	287	130	381	102	85	75	43
8	0	207	2,426	1,005	398	1,164	364	457	227	138
16	0	515	10,050	2,279	1,074	2,936	1,443	1,152	703	460
32	0	1,202	33,065	5,923	2,688	7,228	5,390	3,230	2,183	1,390
average	0	403	9,238	1,904	865	2,362	1,463	987	640	408



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	183	33	12	155	109	31	42	60	65
4	0	472	195	124	459	325	142	237	189	294
8	0	1,195	505	395	1,049	1,298	495	983	656	790
16	0	2,282	1,709	818	2,157	4,122	1,298	2,649	1,874	2,169
32	0	4,731	4,939	2,041	4,720	12,068	3,272	6,118	6,527	6,246
average	0	1,773	1,476	678	1,708	3,584	1,048	2,006	1,861	1,913



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	8.13	1.51	0.36	3.00	0.84	0.25	0.14	0.10	0.10	0.08
4	11.76	4.87	1.42	1.49	2.03	0.44	0.20	0.70	0.19	0.25
8	23.15	8.79	0.81	2.44	2.80	0.77	0.30	0.33	0.29	1.37
16	19.53	10.41	0.90	1.77	3.16	1.21	0.41	0.47	0.40	1.48
32	29.19	11.47	1.23	1.82	3.78	3.06	0.52	0.75	0.54	4.83
average	18.35	7.41	0.94	2.10	2.52	1.15	0.31	0.47	0.30	1.60



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	23.39	20.42	16.09	21.22	19.24	19.37	19.67	18.44	17.93	20.04
4	24.61	20.58	18.03	19.94	21.11	19.38	19.38	20.00	18.57	20.81
8	30.03	25.68	18.68	19.17	20.96	20.40	19.36	19.16	18.04	18.96
16	30.95	23.92	18.76	18.95	20.41	19.74	19.21	18.64	19.40	20.16
32	34.42	24.10	20.11	19.78	21.26	21.19	19.60	18.33	19.80	19.93
average	28.68	22.94	18.33	19.81	20.60	20.02	19.44	18.91	18.75	19.98

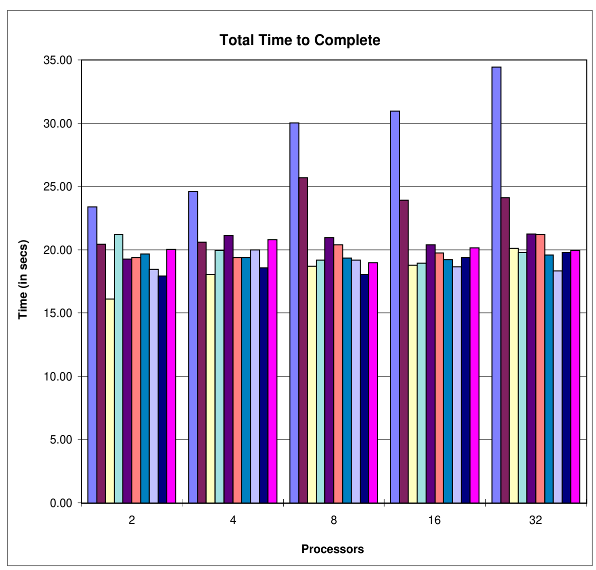
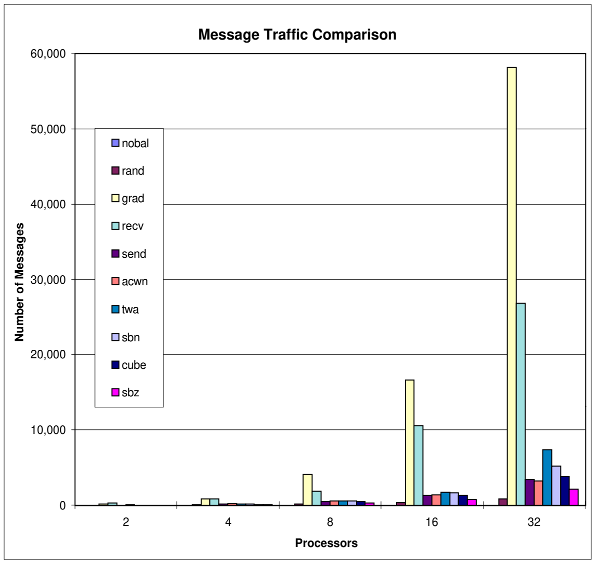
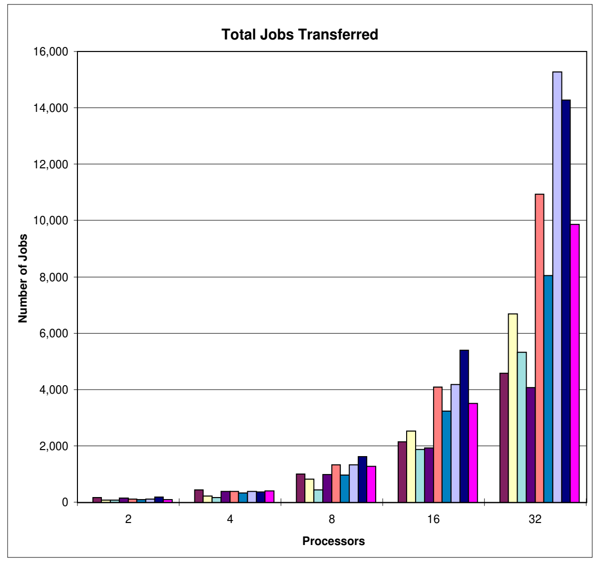


Figure 11. Performance under heavy system load.

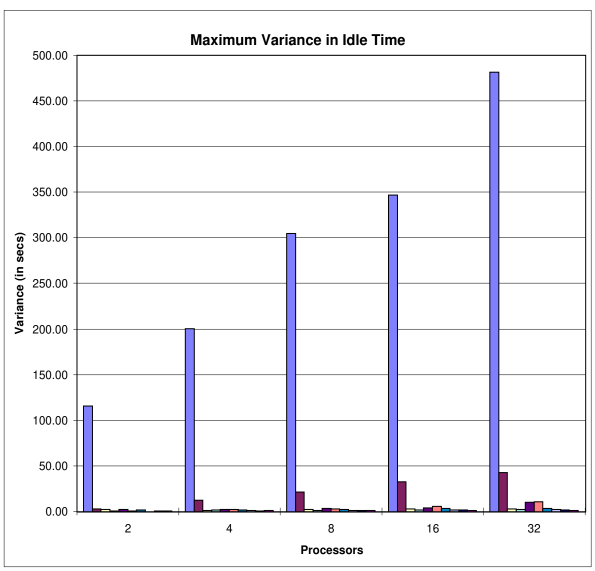
P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	14	139	274	33	48	21	22	31	16
4	0	45	815	848	163	178	121	137	90	95
8	0	147	4,095	1,863	447	534	538	519	495	247
16	0	351	16,601	10,537	1,318	1,367	1,730	1,648	1,326	729
32	0	834	58,155	26,821	3,428	3,216	7,375	5,185	3,824	2,128
average	0	278	15,961	8,069	1,078	1,069	1,957	1,502	1,153	643



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	167	77	76	146	100	87	110	180	99
4	0	438	219	166	373	385	333	385	372	392
8	0	1,004	810	442	976	1,329	956	1,334	1,625	1,272
16	0	2,136	2,520	1,863	1,930	4,085	3,231	4,179	5,399	3,500
32	0	4,574	6,683	5,320	4,077	10,933	8,043	15,271	14,278	9,863
average	0	1,664	2,062	1,573	1,500	3,366	2,530	4,256	4,371	3,025



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	115.55	2.96	2.21	0.30	2.29	0.58	1.47	0.25	0.38	0.48
4	200.23	12.51	1.19	1.65	2.47	2.44	1.51	1.16	0.71	0.89
8	304.77	21.41	2.06	1.34	3.51	2.65	2.41	1.21	1.07	1.03
16	347.08	32.62	2.84	1.82	3.87	5.41	3.24	1.63	1.42	1.10
32	481.26	42.38	2.68	2.26	10.11	10.46	3.21	2.26	1.41	1.12
average	289.78	22.38	2.20	1.47	4.45	4.31	2.37	1.30	1.00	0.92



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	276.28	41.02	41.21	40.86	42.15	40.72	43.38	43.20	40.02	39.90
4	329.78	42.15	40.45	41.39	40.01	39.95	39.97	40.35	40.94	39.90
8	371.73	46.20	40.17	41.11	40.72	40.71	39.97	40.22	40.19	40.00
16	393.45	52.00	39.91	39.96	40.09	40.04	39.94	40.08	39.98	39.92
32	494.20	60.41	39.99	40.09	42.14	40.75	40.09	39.93	39.98	39.91
average	373.09	48.36	40.35	40.68	41.02	40.43	40.67	40.76	40.22	39.93

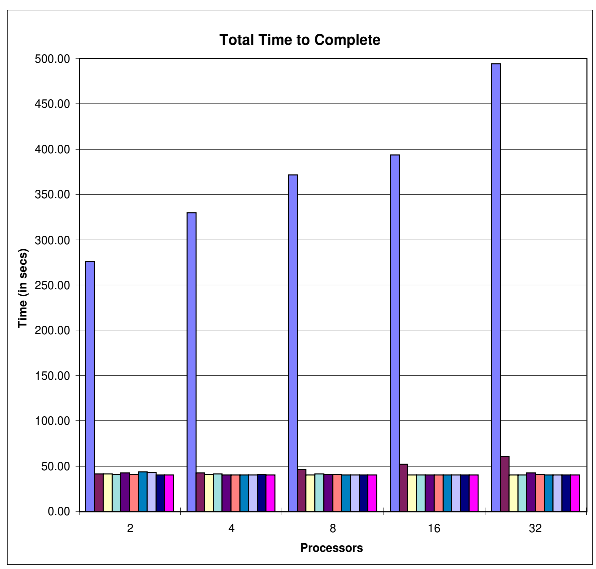
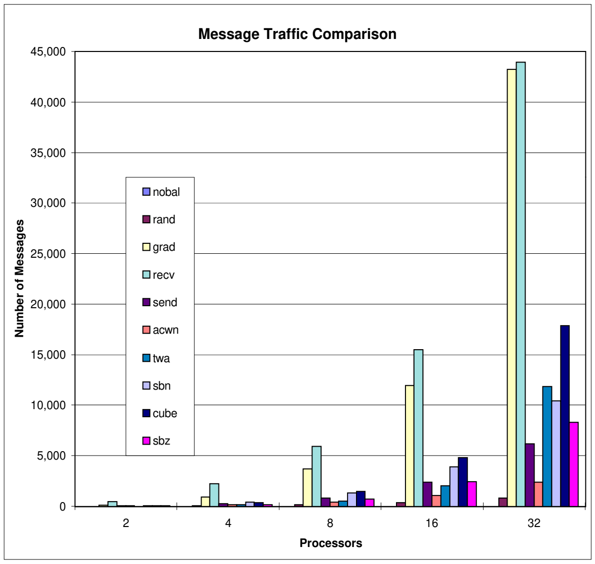
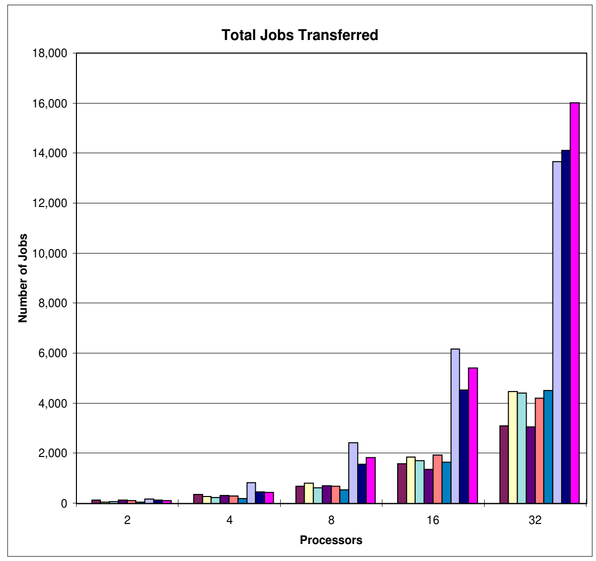


Figure 12. Performance when transitioning from heavy to light system load.

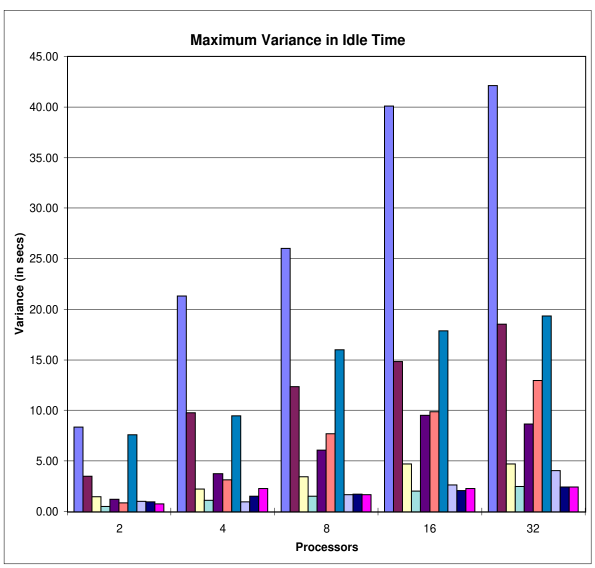
P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	14	87	466	46	42	24	47	60	27
4	0	47	898	2,235	235	149	127	401	371	148
8	0	138	3,718	5,930	805	404	500	1,336	1,462	712
16	0	337	11,921	15,501	2,395	1,045	2,026	3,917	4,810	2,433
32	0	805	43,243	43,950	6,151	2,400	11,862	10,414	17,881	8,313
average	0	268	11,973	13,616	1,926	808	2,908	3,223	4,917	2,327



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	127	47	54	120	101	47	160	129	98
4	0	338	262	218	304	291	194	827	455	422
8	0	676	796	624	689	672	535	2,420	1,549	1,821
16	0	1,580	1,851	1,693	1,346	1,929	1,645	6,172	4,523	5,405
32	0	3,093	4,454	4,411	3,044	4,197	4,509	13,668	14,118	16,007
average	0	1,163	1,482	1,400	1,101	1,438	1,386	4,649	4,155	4,751



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	8.35	3.51	1.49	0.52	1.21	0.85	7.60	1.02	0.98	0.78
4	21.29	9.78	2.21	1.13	3.77	3.13	9.48	0.94	1.50	2.30
8	26.02	12.33	3.42	1.52	6.07	7.67	15.98	1.69	1.74	1.65
16	40.10	14.85	4.71	2.02	9.50	9.89	17.89	2.61	2.09	2.28
32	42.11	18.52	4.73	2.47	8.67	12.94	19.33	4.06	2.44	2.44
average	27.57	11.80	3.31	1.53	5.84	6.90	14.06	2.06	1.75	1.89



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	40.94	39.90	40.11	39.92	40.69	41.01	40.55	39.91	39.96	39.92
4	42.04	40.82	39.91	39.91	40.01	39.93	39.95	39.97	39.92	39.96
8	42.67	39.96	39.93	39.94	40.01	39.92	42.48	39.89	39.97	39.93
16	47.73	39.98	39.97	39.96	40.01	39.95	40.70	39.90	39.96	39.97
32	49.05	39.95	39.96	39.95	40.02	39.91	41.54	39.95	39.97	39.98
average	44.49	40.12	39.98	39.94	40.15	40.14	41.04	39.92	39.96	39.95

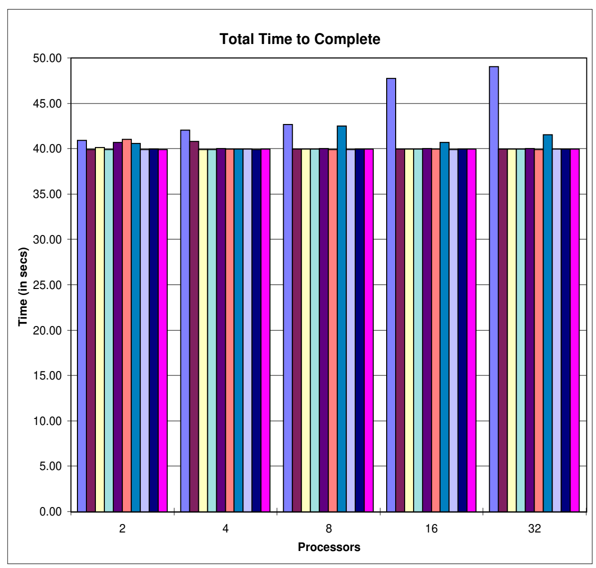


Figure 13. Performance under light system load.

figure 13,) only NOBAL and TWA fail to complete within the optimal amount of time (40.0 secs.) This is consistent with the results just discussed where both approaches show a large variance in processor idle time. Similarly, under the heavy to light load test (see figure 12), most of the algorithms finish at near-optimal times (approximately 40.0 secs). Here, only NOBAL and RAND could not process the job queues within the expected time. If the average values are observed from the chart in figure 11 for the Total Time to Complete, the best average results for the heavy system load test were recorded by SBN, CUBE, and the Gradient (GRAD) algorithms.

Next consider the Message Traffic Comparison metric. As expected, GRAD generates, by far, the largest amount of message traffic. The Receiver Initiated algorithm (RECV) generates a large number of messages because of its tendency to become unstable under light system loads. Idle processors can flood the system with job request messages in situations where their neighbors do not have excess jobs to return. To alleviate this condition, a 0.1 sec delay is introduced between job requests. Longer delays tend to reduce the load balancing effectiveness of RECV under light loads. As expected, SBZ generates less message traffic than the other SBN variants. Likewise, all SBN-based algorithms incur less message passing than TWA. In general, the algorithms that do the worst in terms of load balancing require little or no message communication. For example, NOBAL incurs no message communication, proving that a degree of balancing messages is necessary to balance a system load.

The last metric to consider is Total Jobs Transferred. During the heavy system load test (see figure 11,) the Total Jobs Transferred is significantly less than in the other experiments (see figures 12 and 13). This is due to the fact that during heavy loads, most processors are busy and not seeking additional jobs to process. Under the light system load test (see figure 13,) the situation changes. Here, the SBN-based algorithms generate the largest values because of their tendency to pass jobs through more processors to satisfy those with low loads. It is important to note that the data associated with jobs need to be transferred only once, just before a job is about to execute. Note also that the SBN algorithms utilize bulk transfers in sending distribution messages to relocate jobs. These characteristics reduce the negative effects of message latency and minimize the additional overhead that is incurred. Two of the algorithms (RANDOM and SEND) that only allow one job transfer before execution have the smallest Total Jobs Transferred values, as would be expected. However, an interesting result is that ACWN, which also does not allow jobs to be rerouted more than once, has higher Total Jobs Transferred than the other algorithms. This is due to ACWN's characteristic of doing more work to initially balance the load when the jobs are generated. In fact, during the heavy system load test (see figure 11,) Total Jobs Transferred for ACWN is among the highest.

To evaluate the overall performance of the SBN-based algorithms as compared to the other existing approaches, first note that the performance of CUBE is very similar to that of SBN. This indicates that the comparisons are fair even though the topology of SBN is different from that of the other algorithms. The performance similarity between CUBE and SBN is not surprising since both topologies have a depth of $\log P$ and use the same basic balancing approach.

To continue the overall performance evaluation, let us consider the heavy system load (see figure 11) experiment. In this test, Total Time to Complete is the most important metric. Clearly, NOBAL and RAND are the worst performers, and are hence non-competitive. Looking at the average performance numbers for the experiment, it is found that the SEND, ACWN, and SBZ strategies are worse than the average performance of the SBN and CUBE algorithms by 6% to 10%.

Next consider the experimental results under the transition from heavy to light system load (see figure 12) and the light system load (see figure 13) scenarios for the remaining five algorithms (GRAD, RECV, TWA, SBN, and CUBE.) In these tests, Message Traffic Comparison is the most important metric since it is expected that the completion times will be near-optimal for all reasonable load balancing approaches. Figures 12 and 13 show that GRAD and RECV have significantly greater message traffic than TWA and the SBN-based algorithms.

In comparing TWA to the SBN algorithms, it is also desirable to determine how much load balancing overhead is incurred by TWA. This is important because TWA suspends application processing during balancing. The results show that TWA spends between 3.23% and 4.86% of the execution time in balancing a network of 32 processors. With a network of 16 processors, the overhead varies between 0.73% and 1.08%. Based on the superlinear increase in the fraction of time spent by TWA in balancing the load, the overhead could potentially become intolerable for large networks of processors. By contrast, SBN hides all of the load balancing time since processing is never suspended.

Based on the forgoing analysis that compares the SBN-based load algorithms to other balancing schemes, it was shown that the SBN approach is an effective load balancing alternative and compares favorably. Intuitively, the SBN approach is effective because global load information is obtained from all processors to balance the system load. Most of the existing load balancing algorithms work only locally with processors interacting with their immediate neighbors. Therefore, the load information is likely to be less accurate. Even though the SBN approach is global, the load balancing overhead is acceptable because the depth of the SBN is $\log P$, where P is the number of available processors. The fact that SBN allows processing to continue during load balancing hides communication and data distribution costs because application processing is not interrupted.

3.5 Summary

In this chapter, three effective load balancing algorithms were proposed (Basic, Hypercube Variant, and Heuristic Variant) based on a topology-independent logical communication pattern among processors, called Symmetric Broadcast Networks (SBN). This approach to load balancing has been shown to compare favorably to several other schemes. The metrics that measure the Maximum Variance in Idle Time and Total Time to Complete, demonstrate that all three proposed algorithms are effective in balancing the system load while minimizing the idle time. The Message Traffic Comparison metric shows that use of the Heuristic Variant reduces overhead associated with load balancing traffic when compared to the two other SBN-based algorithms.

CHAPTER 4

MESH ADAPTATION WITH SBN ALGORITHMS

The ability to dynamically adapt an unstructured grid (or mesh) is a powerful tool for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult, particularly from the viewpoint of portability on various multiprocessor platforms.

Numerical solution schemes for problems in computational science and engineering are usually performed on a mesh of vertices and edges. These applications can be modeled in the load balancing framework as a graph of jobs where adjacent jobs need to communicate in order to complete their tasks. The traditional approach to such problems is to find a near-optimal minimal cut that partitions the graph among the processors in a network while balancing the individual processor workloads. For applications that require dynamic mesh adaptation, the load balancing procedure has to be invoked whenever the system load becomes unbalanced. Load balancing involves both partitioning the computational mesh and mapping the resulting partitions to the processors. Mapping requires data to be redistributed, that is, moved from one processor to another as determined by the partitioner. An example of this semi-dynamic approach is the PLUM [60] framework which provides an automatic portable (i.e., architecture-independent) environment for performing adaptive numerical computations in a message-passing environment. PLUM applies an efficient algorithm to the output mapping of a partitioner to minimize the reassignment overhead. Any general-purpose partitioner can be used by PLUM.

The disadvantage of the semi-dynamic approach is that application processing is temporarily suspended while the load is balanced. In contrast, dynamic load balancers such as those that are SBN-based, allow application processing to continue while load balancing takes place. This feature presents opportunities for latency tolerance techniques that will allow processing and communication/remapping costs to be overlapped. These advantages present opportunities for increased efficiency and reduced overall runtime requirements.

The SBN approach as detailed in chapter 3 is based on defining a robust communication pattern (logical or physical) among processors, called *Symmetric Broadcast Networks*. The experiments presented in section 3.4 with synthetic loads [18] have demonstrated that an SBN-based load balancing solution achieves excellent performance when compared to other popular general purpose balancing techniques such as Random, Gradient, Receiver Initiated, Sender Initiated, and Adaptive Contracting. The new application-based SBN load balancer is modified in this chapter to provide a global view of system loads across processors and demonstrate that a dynamic load balancer can be a viable choice to solve adaptive unstructured mesh applications. This adaptive, and decentralized scheme can be ported to any topological architecture through efficient embedding techniques. Experiments on an IBM SP2 and an SGI Origin2000 show that the SBN-based balancer achieves superb load balance with minimal extra overhead. Experimental results presented in this chapter, however, were obtained from runs performed on an IBM SP2.

4.1 Motivations

The success of parallel computing in solving real-life, computation-intensive problems relies on their efficient mapping and execution on commercially available multiprocessor architectures. When the algorithms and data structures corresponding to these problems are dynamic in nature in the sense that their computational workloads grow or shrink at runtime, or when they are intrinsically unstructured, mapping them onto distributed-memory parallel machines with dynamic load balancing offers considerable challenges. Dynamic load balancing aims to balance processor workloads at runtime while attempting to minimize the communication between processors. A problem is therefore load balanced when processors have nearly equal loads with reduced communication among themselves. With the proliferation of parallel computing, dynamic load balancing has become extremely important in several disciplines like scientific computing, task scheduling, sparse matrix computations, parallel discrete event simulation, and data mining.

The ability to dynamically adapt an unstructured mesh is a powerful tool for efficiently solving computational problems with evolving physical features. Standard fixed-mesh numerical methods can be made more cost-effective by locally refining and coarsening the mesh to capture these phenomena of interest. Unfortunately, an efficient parallelization of these adaptive methods is rather difficult, primarily due to the load imbalance created by the dynamically-changing nonuniform grid. This requires significant communication at runtime, leading to idle processors and adversely affecting the total execution time. Nonetheless, it is generally thought that unstructured adaptive-grid techniques will constitute a significant fraction of future high-performance supercomputing.

As an example, if a full-scale problem in computational fluid dynamics were to be solved efficiently in parallel, dynamic mesh adaptation would cause load imbalance among processors. This, in turn, would require large amounts of data movement at runtime. It is therefore imperative to have an efficient dynamic load balancing mechanism as part of the solution procedure. However, since the computational mesh will be frequently adapted for unsteady flows, the runtime load also has to be balanced at each step. In other words, the dynamic load balancing procedure itself must not pose a major overhead. Although several dynamic load balancers were proposed for multiprocessor platforms [12, 15, 18, 42, 51, 75, 76], most of them are inadequate for adaptive unstructured grid applications because they lack a global view of system loads across processors. Also, workload migration in these approaches does not take into account the structure of the adaptive grid. This motivates the work presented in this chapter.

4.2 SBN-based balancer modifications

Execution time, communication cost, and data distribution overhead are all used to determine how long a given job will take to complete where the time required to execute a given job is determined by actual load data. Before the modifications to the Basic SBN algorithm can be described, the following terms need to be defined:

- Wgt^j denotes the computational cost to process job j.

- $Comm_p^j$, is the communication cost associated with job j executing on processor p. Thus, $Comm_p^j$ is calculated as:

$$Comm_p^j = \sum_{i=1}^{\alpha} R_i \times \mathtt{CWgt}_{(j,i)}$$

  where R_i is the fraction of j's execution time that the data for adjacent job i resides at a remote processor, α is the number of adjacent jobs, and $\mathtt{CWgt}_{(j,i)}$ is the communication cost between jobs j and i. Note that job completion requires $\mathtt{CWgt}_{(j,i)}$ extra units of time for inter-job communication, if job i's data set resides at a remote processor during the entire run.

- $Remap_p^j$ is the redistribution cost to copy the data set for job j to processor p from another processor. If a data set does not need to be migrated, $Remap_p^j = 0$.

The major modifications to the Basic SBN algorithm that take these factors into account and provide a global view of the adaptive mesh can now be described.

- For each processor p, a weighted system load level, $\texttt{WSysLL}$, and weighted queue length, $\texttt{QWgt}(p)$, are used rather than determining the system load based solely on the queue length (e.g., $\texttt{SysLL}$ and $\texttt{QLen}(p)$ described in chapter 3.) These values account for the total processing, communication, and data distribution times required to run the jobs in the system to completion. For the sake of the experiments that we present in this dissertation, the units of cost associated with execution time, communication, and data set remapping are assumed to be equal.

  The formulae used for maintaining $\texttt{QWgt}(p)$ and $\texttt{WSysLL}$ are:

$$\texttt{QWgt}(p) = \sum_{j \text{ assigned to } p} (Wgt^j + Comm_p^j + Remap_p^j),$$

  and

$$\texttt{WSysLL} = \left\lceil \frac{1}{P} \sum_{i=1}^{P} \texttt{QWgt}(i) \right\rceil.$$

- The minimum and maximum thresholds, $\texttt{MinTh}$ and $\texttt{MaxTh}$ respectively, were altered to reflect the time passed when all locally-queued jobs are processed. Also, in order to prevent excessive job distributions, a limit was placed on the number of jobs that can be migrated at once.

- The SBN-based algorithm utilizes multiple heaps (or priority queues) to decide which jobs to process and which jobs to migrate. All jobs queued for processing are stored in a single local "processing heap" and in one of a group of local "migration heaps." One local migration heap exists for each processor in the network. Table 1 illustrates a group of jobs queued for processing at processor zero. The corresponding processor locations of the data sets for these jobs are also shown. For example, jobs one and two are on migration heap zero, while jobs three, four, and five are stored on migration heaps two, three, and one, respectively. Note that the migration heap in which a job is placed corresponds to the processor at which the job's data set resides. By using this data structure, the SBN algorithm can quickly favor migration of jobs that are to be moved to the processor where their data are stored.

Table 1. An example of jobs queued for execution at processor zero

Job #	Data on Processor #
1	0
2	0
3	2
4	3
5	1

For the example in table 1, if a job were to be distributed to processor 3, job 4 would be favored over others since its data set is stored on processor 3. The balancing algorithm can quickly remove that job from the corresponding migration heap. Also, the algorithm can efficiently choose jobs to migrate that would incur a minimum increase in the system distribution and communication costs when the job is run. The heap order reflects this calculation. Lastly, the processing heap is used to order jobs to be processed that would be the most expensive to migrate.

- Migration messages were added to simulate the movement of data sets from one processor to another. Using the SBN, a migration message is broadcast when a job is about to be run and its corresponding data set is not resident. A locate message is then broadcast to indicate the new location of the data set. Therefore, all processors can maintain the processor location of all data sets.

4.3 SBN adaptive mesh experiment

In this chapter, a computational mesh is used that is generated from the simulation of an acoustics experiment where a one seventh-scale model of a UH-1H helicopter rotor blade was tested over a range of sub-sonic and transonic hover-tip Mach numbers. The initial mesh consisted of 13,967 vertices, 60,968 tetrahedral elements, and 78,343 edges. A cut-out view of the initial tetrahedral mesh is shown in figure 14. In this application, varying fractions of the domain are targeted for refinement based on an error indicator calculated directly from the flow solution.

Numerical results and a detailed report of the simulation are available in [9, 73]. A total of three adaptations were performed on this initial mesh using a solution-based dynamic mesh adaptation procedure [8]. The final mesh contained 137,474 vertices, 765,855 tetrahedra, and 913,412 edges. However, load balancing was always performed on the initial mesh by modifying the vertex weights of the corresponding dual graph to model the dynamic mesh adaptation [60].

With the modified SBN algorithms, two sets of experiments are executed using this dynamic mesh adaptation load data. In the first experiment, the migration of jobs with long execution times was favored. These results are presented in table 2. In the second experiment, jobs with short execution times were given the higher priority for migration. The corresponding results are presented in table 3. Results obtained from the presented experiments on real data show that the SBN approach is an effective load balancing technique.

The column headings for tables 2 and 3 are defined as follows:

P: Number of processors.

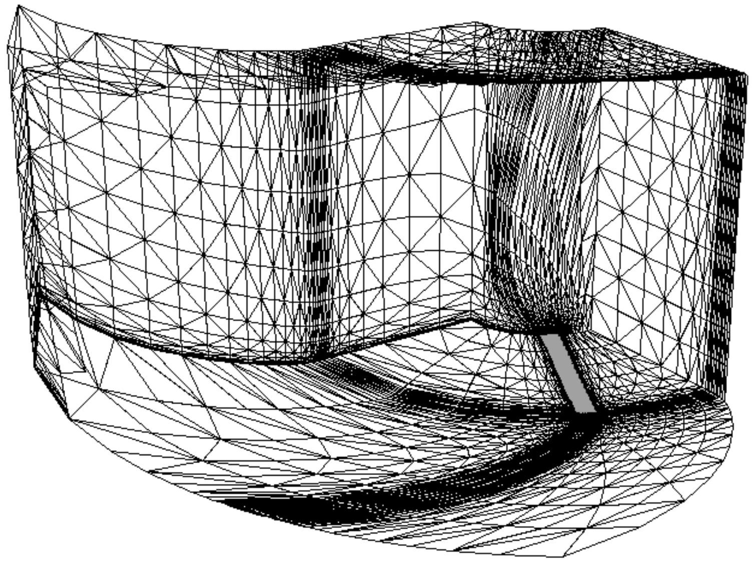


Figure 14. Cut-out view of the initial tetrahedral mesh.

Jobs Transferred: Total number of jobs migrated from one processor to another.

Total Messages: Total number of load balancing, distribution, and job-migration messages that are processed during the experiment.

Processing Percentage: Average percentage of total time each processor spends executing the application (simulated using the Wgt^j values.)

Communication Percentage: Average percentage of total time each processor spends in communication routines (simulated using $Comm_p^j$ values.) Note that communication messages are not actually transmitted during the experiment.

Distribution Percentage: Average percentage of total time each processor spends in routines that distribute jobs (simulated using $Remap_p^j$ values.) Recall that actual mesh data sets are not transmitted during the experiment.

Idle Percentage: Average percentage of total time that each processor is idle.

Table 2. Adaptive mesh experiment favoring long job migration

P	Jobs Transferred	Total Messages	Processing Percent	Communication Percent	Distribution Percent	Idle Percent
2	3,567	9,068	92.85	1.93	5.18	0.03
4	6,745	76,555	89.27	4.09	6.49	0.15
8	12,767	426,169	84.22	7.15	8.22	0.41
16	128,780	2,759,716	79.56	11.28	8.45	0.70
32	1,261,500	15,915,048	76.05	11.51	11.11	1.33
64	10,854,794	96,184,273	72.25	11.53	13.33	2.89

4.4 Summary

Results show that both experiments balance the load in such a way that minimal idle time is achieved. Favoring migration of large jobs increases the percentage of time spent processing jobs and lowers the number of jobs migrated. However, this improvement requires significantly more message traffic. If these results are extrapolated, minimal changes in idle time percentage occur as the number of processors are increased. Both experiments demonstrate that the SBN approach is general enough to be applicable to this class of problems.

It was also demonstrated that the SBN approach, can be effectively applied to a dynamic adaptive-mesh application to balance the load among processors while at the same time minimizing the required communication and distribution costs. The SBN approach provides a global view of the adaptive mesh, and thereby overcomes many of the drawbacks associated with traditional dynamic load balancers.

Table 3. Adaptive mesh experiment favoring short job migration

P	Jobs Transferred	Total Messages	Processing Percent	Communication Percent	Distribution Percent	Idle Percent
2	29,298	58,531	82.89	4.05	13.04	0.02
4	53,636	80,991	80.15	7.79	12.03	0.03
8	84,225	514,549	76.15	12.10	11.63	0.12
16	182,036	2,074,743	72.80	15.92	10.77	0.51
32	913,601	7,139,339	70.87	16.62	11.59	0.92
64	3,603,769	19,858,167	68.32	17.46	12.88	1.34

CHAPTER 5

SBN-BASED ALGORITHM COMPARISONS

In this chapter, the SBN-based load balancing algorithms presented in chapter 4 are further modified. The goal of these changes is to make better use of the global view of system loads across processors and implement metrics that can be used to draw further conclusions. Results presented (i) demonstrate the portability of the SBN approach with experiments performed on an IBM SP2 and SGI Origin2000 without any required programming modifications; (ii) draw comparisons between the SBN dynamic balancer and the PLUM semi-dynamic framework; and (iii) combine the dynamic and semi-dynamic load balancing approaches to determine if additional benefits can be realized. Results show that both the dynamic and semi-dynamic load balancing approaches achieve superb load balance with minimal extra overhead. The SBN algorithms reduce the cost of data set redistribution; however, a higher communication cost during processing is incurred.

In section 5.1, for sake of completeness, the PLUM semi-dynamic framework is presented for solving computational mesh applications. Background information describing a dual graph representation used in connection with balancing the load is also discussed. Next, section 5.2 describes refinements to the SBN mesh balancing algorithms and contrasts the approach with PLUM. Section 5.3 describes the computational test case and introduces established metrics that will be used to draw conclusions. An SBN overhead analysis is presented in section 5.4 and comparisons with PLUM are made in section 5.5. Section 5.6 experiments with combining the dynamic and semi-dynamic approaches and draws some additional conclusions. Finally, section 5.7 mathematically analyzes the SBN mesh balancing algorithms.

5.1 PLUM semi-dynamic framework

PLUM is an automatic and portable load balancing environment, specifically created to handle adaptive unstructured grid applications. It differs from most other similar load balancers in that it dynamically balances processor workloads with a *global* view. Prior work [7, 60] has successfully demonstrated the viability and verified the effectiveness of PLUM for various test cases involving adaptive unstructured grids. In [20] its architecture-independent features are established by successfully performing mesh adaptations on an IBM SP2, an SGI Origin2000, and a CRAY T3E without any code modifications. Preliminary versions of these results appear in [60, 61].

A major improvement of the PLUM framework over previous solvers is its efficient algorithm for minimizing this remapping overhead to guarantee an optimal processor reassignment. This algorithm is applied to the default output mapping of a parallel partitioner and significantly reduces the reassignment overhead.

Figure 15 provides an overview of PLUM. After an initial partitioning and mapping of the unstructured grid, a `Solver` executes several iterations of the application. A mesh adaptation procedure is invoked when the computational mesh needs to be refined or coarsened. Mesh edges are targeted for coarsening or refinement based on an error indicator computed from the solution. The old mesh is then coarsened where possible, resulting initially in a smaller grid. This coarsening reduces the overhead associated with the partitioning step to follow.

Since edges were already marked for refinement, the new mesh can be exactly predicted before actually performing the refinement step. Program control is passed to the load balancer at this time. PLUM then gains control to determine if the workload among the processors has become unbalanced due to the mesh adaptation, and to take appropriate action. Specifically, if the load has become unbalanced, the adapted mesh is repartitioned to divide the new mesh into submeshes. PLUM then utilizes an efficient heuristic algorithm to minimize the remapping overhead by guaranteeing an optimal processor reassignment. The data distribution cost is significantly reduced by applying this heuristic to the output mapping generated by the parallel partitioner. Therefore, the repartitioned mesh is efficiently distributed among the processors.

If the remapping cost is compensated by the computational gain that would be achieved with balanced partitions, all necessary data are appropriately redistributed. Otherwise, the new partitioning is discarded and execution continues without remapping. After the grid data sets are remapped among the processors, the computational mesh is refined and the numerical calculation is restarted.

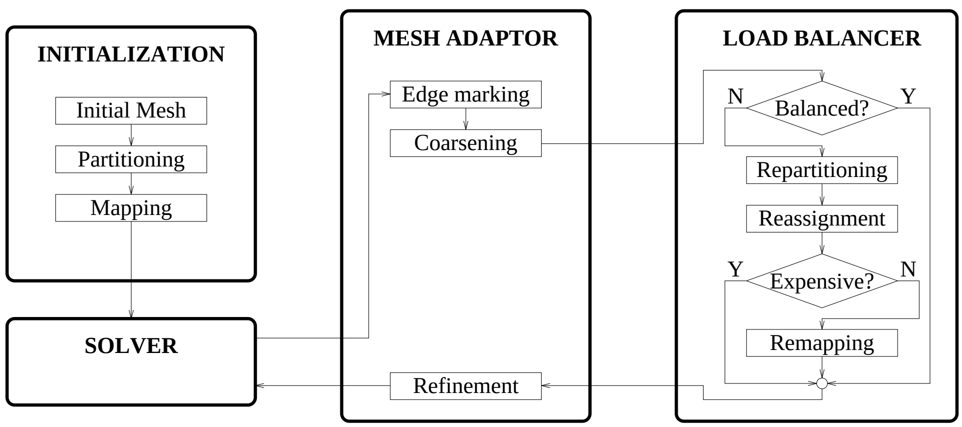


Figure 15. Overview of the PLUM environment.

The PLUM load balancer features (i) repeated use of the initial (mesh) dual graph during the course of an adaptive computation, (ii) parallel mesh repartitioning, and (iii) an efficient remapping and data movement scheme. For the sake of completeness, a brief overview of these salient features are discussed in subsections 5.1.1, 5.1.2, and 5.1.3.

5.1.1 Repeated initial dual graph use

Mesh adaptation problems often represent three dimensional space using a grid of mesh elements such as hexahedra or tetrahedra. As portions of the mesh are coarsened or refined to capture areas of interest, these elements are typically divided into two, four, or eight sub-elements. An example of a tetrahedral division into two sub-elements is shown in figure 16. A series of mesh refinements result in a tree of these elements being formed. In figure 16, the refinement tree has three elements (the original element and two children shown on the left.) For load balancing, it is important to construct a graph structure that can be used for partitioning. It is also important to be able to make use of the same graph regardless of how many mesh refinements have been executed. This is accomplished by both PLUM as well as by SBN using the dual graph of the three dimensional mesh. Namely, each vertex in the dual graph corresponds to an *original* mesh element. The vertices of the dual graph have two weights, $\texttt{PWgt}_v$ and $\texttt{RWgt}_v$, and each edge of the graph has one weight, $\texttt{CWgt}_{(v,w)}$. These weights respectively represent the cost (in units) associated with processing, data set relocation, and communication between mesh elements. On the right side of figure 16, these costs are depicted as P, R, and C respectively.[1] $\texttt{PWgt}_v$ is the number of leaf elements in the refinement tree (two in figure 16 after refinement) because the leaf elements correspond to the number of elements that need to be processed in a particular adaptation step. Similarly, $\texttt{RWgt}_v$ is the total number of elements in the refinement tree (three in figure 16 after refinement) because all of the elements need to be relocated should a mesh element be moved from one processor to another. Finally, $\texttt{CWgt}_{(v,w)}$ equals the number of common faces between adjacent elements (two or one in figure 16 after refinement) because that weight represents the amount of communication required between those adjacent elements. The advantage

[1]Note that the same vertex is shown both before (left) and after (right) refinement on the right side of figure 16.

of this dual graph representation is that the same graph can be used by changing weights regardless of the number of refinements. The dual graph is also a convenient structure that can be presented to a partitioner or to SBN to accomplish load balancing operations.

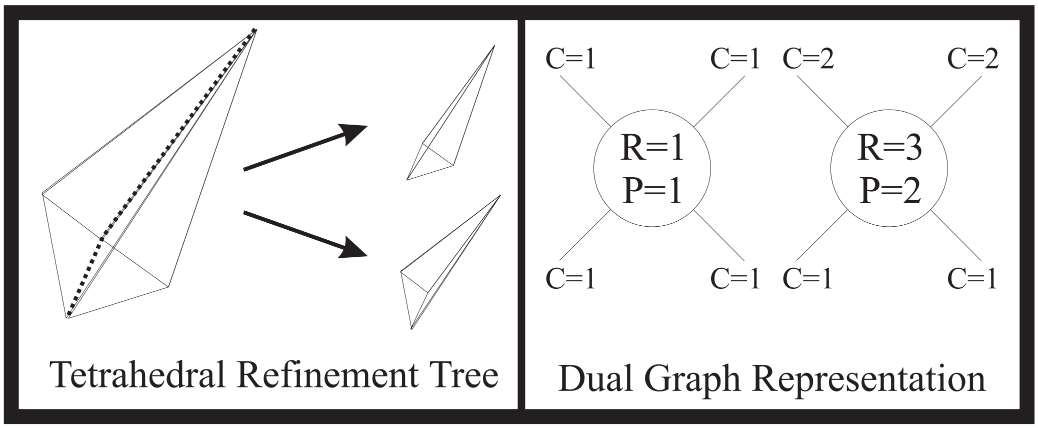


Figure 16. Dual graph mesh representation.

5.1.2 Parallel mesh repartitioning

As the computational mesh is adapted and the processor workloads become unbalanced, the mesh needs to be repartitioned. PLUM can use any general-purpose partitioner that balances the computational loads and minimizes the runtime interprocessor communication. However for PLUM to be viable, the repartitioning must be performed rapidly, and must not result in large shifts in data that would increase the remapping and communication overhead.

The overall effectiveness of repartitioning algorithms is determined by how successful they are in load balancing the computations while minimizing the edge-cut, as well as the cost associated with redistributing the load in order to realize the new partitioning. Data redistribution is generally considered the most expensive phase in dynamic load balancing. The migration of mesh objects requires a number of costs such as the communication overhead of remote-memory latency, and the computational overhead of rebuilding internal and shared data structures. Since global partitioners do not consider the redistribution cost when generating subdomains, an intelligent algorithm is needed to map the new partitions onto the processors such that the data migration cost is minimized. Diffusive repartitioners explicitly attempt to minimize the data redistribution cost when generating new subdomains by using the current partition as a starting point. These strategies are generally successful when there are gradual changes in the load distribution. However, diffusive schemes may not be well suited for problems which incur dramatic shifts in the load imbalance between redistributions. Experimental results [7] indicate that for applications of this kind, the data migration overhead can be reduced by combining diffusive repartitioners with intelligent remapping algorithms.

Since graph partitioning is an NP-complete problem, research has focused on developing efficient heuristic algorithms. Many graph partitioning algorithms were developed over the years, particularly for static grids. Significant progress was made in improving the partitioning heuristics as well as generating high-quality software. Several excellent parallel partitioning algorithms are now available [30, 45, 77, 46]. Among the partitioning approaches are simulated annealing [49], genetic algorithms [48], clustering [29, 11], spectral [41] and geometry-based [71, 62].

Multilevel algorithms reduce partitioning times considerably by contracting the graph, partitioning the coarsened graph, and then refining it to obtain a partition for the original graph. MeTiS [45] and Jostle [77] are currently the fastest multilevel schemes that are available. Both MeTiS and Jostle are available in various flavors, and can be run either as partitioners from scratch or as diffusive repartitioners which modify initial partitions that are provided.

ParMeTiS [44] is a state-of-the-art global partitioner that makes no assumptions on how the mesh is initially distributed among the processors. It is a k-way multilevel algorithm that operates in three phases: (i) in the initial coarsening phase, the original mesh, M_0, is reduced by collapsing adjacent vertices or edges through a series of smaller and smaller meshes to M_k, such that M_k has a sufficiently small number of vertices; (ii) in the partitioning phase, the workload is balanced among the processors by using a greedy graph bisection algorithm; and (iii) in the projection phase, the partitioned mesh M_k is gradually restored to its original size M_0, while using a variant of the Kernighan-Lin refinement algorithm [47] to minimize the edge cut size. This is the strategy that was chosen for implementation within the PLUM framework.

5.1.3 Data set reassignment

The goal of processor reassignment is to find a mapping between partitions and processors that minimize the cost of data set redistribution. PLUM utilizes an efficient heuristic algorithm to reassign vertices among processors. The idea behind the algorithm is to construct a similarity matrix that enables a near optimal processor mapping to be efficiently computed that minimizes data movement with respect to either the `TotalV`, `MaxV`, or `MaxSR` metrics. `TotalV` minimizes the total volume of data moved among all the processors; `MaxV` minimizes the maximum flow of data to *or* from any single processor, while `MaxSR` minimizes the sum of the maximum flow of data to *and* from any processor. Key comparisons between SBN and PLUM in this chapter are based on the `MaxSR` metric. The PLUM heuristic data set reassignment algorithm is presented in greater detail in [6] and [60].

5.2 The SBN dynamic approach

When selecting vertices to be processed, the SBN load balancer takes advantage of the underlying structure of the adaptive grid and defers local execution of boundary vertices as long as possible because they may be migrated for more efficient execution. Thus, the next queued vertex is selected such that it minimizes the overall cut size of the adapted grid. A priority min-queue is maintained for this purpose, where the priority of a vertex v in processor p is given by $(Comm_p^v + Remap_p^v)/Wgt^v$. Therefore, vertices with no communication and redistribution costs are executed first. Those with low communication or redistribution overhead relative to their computational weight are processed next. Conceptually, internal vertices are processed before those on partition boundaries.

Additionally, the SBN-based balancer, reduces runtime communication and the cost of moving data sets through efficiently choosing which vertices to reassign when the processing load is balanced. The differential edge cut, described in the next section, is used to accomplish these goals.

5.2.1 Differential edge cut

For balancing the system load among processors, an optimal policy for vertex migration needs to be established. When vertices are being moved, assume that processor p is about to reassign some of its vertices to another processor q. The SBN load balancer running on processor p randomly picks a subset of ten vertices from those queued locally. For each selected vertex v, the differential edge cut $\Delta\mathtt{Cut}$, is calculated as follows: [2],

$$\Delta\mathtt{Cut} = Remap_q^v - Remap_p^v + Comm_q^v - Comm_p^v.$$

If $\Delta\mathtt{Cut} > 0$, it is normalized as $\Delta\mathtt{Cut}/Wgt^v$. The parameters $Remap_p^v$ and $Remap_q^v$ will either be zero or equal to the redistribution cost of moving the data for v from p to q. For example, assume $p = 3$, $q = 6$, the data for v reside on $p = 1$, and its redistribution cost is 8. In this case, $Remap_p^v = Remap_q^v = 8$. Similarly, if the data for v resides on $p = 3$, then $Remap_p^v = 0$ but $Remap_q^v = 8$.

A positive $\Delta\mathtt{Cut}$ indicates that an increase in communication and redistribution costs will result if v is migrated from p to q. Therefore, the formula favors migrating

[2]This deviates from the usual definition of edge cut to account for the dynamic SBN algorithm.

vertices with the smallest increase in communication cost per unit computational weight. In contrast, negative Δ`Cut` values indicate a reduction in communication and redistribution costs, hence favoring the migration of vertices with the largest absolute reduction in communication and redistribution costs.

Once Δ`Cut` is calculated for all the randomly chosen vertices, the vertex `MinV` with the smallest value of Δ`Cut` is chosen for migration. Next, following a breadth-first search, the SBN balancer selects the vertices adjacent to `MinV` that are also queued locally for processing at p. The breadth-first search stops either when no adjacent vertices are queued for local processing at p, or if a sufficient number of vertices were found for migration. If more vertices still need to be migrated, another subset of vertices are randomly chosen and the procedure is repeated. This migration policy strives to maintain or improve the cut size during the execution of the load balancing algorithm. In contrast, traditional dynamic load balancing algorithms do not consider cut size and hence are likely to experience larger cut sizes as execution proceeds.

5.2.2 Data redistribution policy

The redistribution of data is performed in a "lazy" manner. Namely, the data set for a given vertex v in a processor p is not moved to q until the latter processor is about to execute v. Furthermore, the data sets of all vertices adjacent to v that are assigned to q are migrated as well. This policy greatly reduces both the redistribution and communication costs by avoiding multiple migrations of data sets and having resident all adjacent vertices that are assigned to processor q while v is being processed.

Data migration is implemented by broadcasting a job migration message when a vertex is about to be processed and its corresponding data set is not resident on the local processor. A locate-message is then broadcast to indicate the new location of the data set. This policy is expected to maximize the number of adjacent vertices that are local when a grid point is processed. Hence, by considering the underlying grid structure, the communication overhead is reduced.

5.2.3 An illustrative example

Figure 17 illustrates the operation of the SBN-based load balancer. It shows a mesh of sixteen vertices and twenty edges that is partitioned among four processors, $P1$ through $P4$. For each vertex, the processing and redistribution costs are represented as a two-tuple within parentheses. For instance, vertex nine has a processing cost of one and a redistribution cost of two. Adjacent vertices on the diagram are connected by edges which are labeled with the associated communication cost. For example, the communication cost between vertices ten and eleven is three, provided the data sets for the two vertices reside on different processors when either one is processed. Each of the four processors are shaded differently in the figure. For example, vertices one, five, six, and nine are assigned to $P3$.

Table 4 shows the Wgt^v, $Comm_p^v$, and $Remap_p^v$ values for each vertex v, under the current vertex-to-processor assignment. Assume that the data for vertex seven are resident in $P2$, the data for vertex ten are in $P3$, while the data for vertices nine, eleven, and sixteen are resident in $P1$. The data sets for the remaining vertices are assumed to be resident in the processor to which the vertices are assigned. Table 4 also shows the $\texttt{QWgt}(p)$ values for each processor p, as defined in section 4.2. The weighted system load $\texttt{WSysLL}$, for this example is twenty four.

Assuming that $\texttt{MinTh}$ is ten, processor $P2$ is clearly underloaded. According to the SBN communication pattern shown in figure 1, $P2$ sends a load balancing request to $P4$. Upon receiving it, $P4$ determines which vertices to migrate to $P2$ so that their loads will be equidistributed. Let us step through the process of selecting the initial vertex to migrate, using the differential edge cut shown in section 4.2. The $\Delta\texttt{Cut}$ values

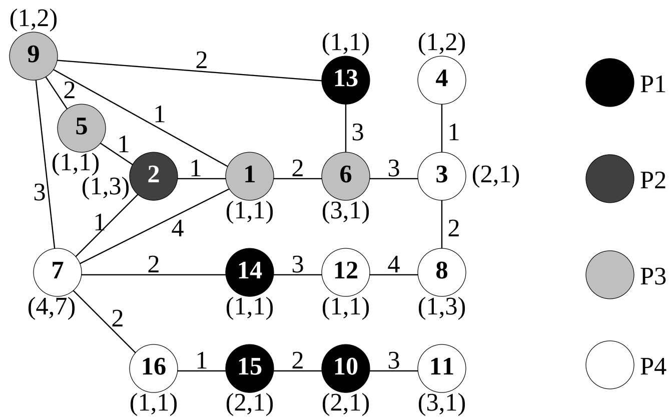


Figure 17. An Illustration of the SBN-based load balancer.

Table 4. Various costs and weighted queue lengths for each vertex v and processor p

processor p	$P1$				$P2$	$P3$				$P4$						
vertex v	10	13	14	15	2	1	5	6	9	3	4	7	8	11	12	16
Wgt^v	2	1	1	2	1	1	1	3	1	2	1	4	1	3	1	1
$Comm_p^v$	0	3	5	2	2	6	3	6	5	3	0	12	0	3	3	3
$Remap_p^v$	1	0	0	0	0	0	0	0	2	0	0	7	0	1	0	1
$\texttt{QWgt}(p)$	17				3	28				46						

of the vertices currently assigned to $P4$ are shown in table 5. Vertex seven is found to be optimal for migration to $P2$, yielding $\texttt{QWgt}(P4) = 23$ and $\texttt{QWgt}(P2) = 18$. The new value of $\texttt{WSysLL} = \lceil 86/4 \rceil$ is 22 which reflects an improvement in the total system load. For this example, additional migration is not necessary.

Table 5. Differential edge cut for each vertex in $P4$ if migrated to $P2$

vertex v	3	4	7	8	11	12	16
$Remap^v_{\mathrm{P2}}$	1	2	0	3	1	1	1
$Remap^v_{\mathrm{P4}}$	0	0	7	0	1	0	1
$Comm^v_{\mathrm{P2}}$	6	1	11	6	3	7	1
$Comm^v_{\mathrm{P4}}$	3	0	12	0	3	3	3
Δ`Cut`	4	3	-8	9	0	5	-2

5.2.4 Differences between SBN and PLUM

The SBN-based load balancer differs from PLUM in several ways:

- Processing is temporarily halted under PLUM while the load is being balanced. During the suspension, a new partitioning is generated and data are redistributed among the nodes of the network. The SBN approach, on the other hand, allows processing to continue while the load is dynamically balanced. This feature also allows for the possibility of utilizing latency-tolerant techniques to hide the communication and redistribution costs during processing.

- Under PLUM, suspension of processing and subsequent repartitioning does not guarantee an improvement in the quality of load balance. If it is determined that the estimated re-mapping cost exceeds the expected computational gain that is achieved by a load balancing operation, processing continues using the original grid assignment. This could result in unnecessary idle time. In contrast, the SBN approach will always result in improved load balance among processors.

- PLUM redistributes all necessary data to the appropriate nodes immediately before processing continues. SBN, however, distributes in a "lazy" manner. Data is migrated to a processor only when it is ready to process the data, thus reducing redistribution and communication overhead.

- The load balancing under PLUM takes place *before* the solver phase of the computation, whereas SBN balances the load *during* the solver execution. One therefore cannot directly compare PLUM and SBN, since their relative performance is solver dependent. However, experiments described in subsection 5.4 makes comparisons between the two approaches using established metrics.

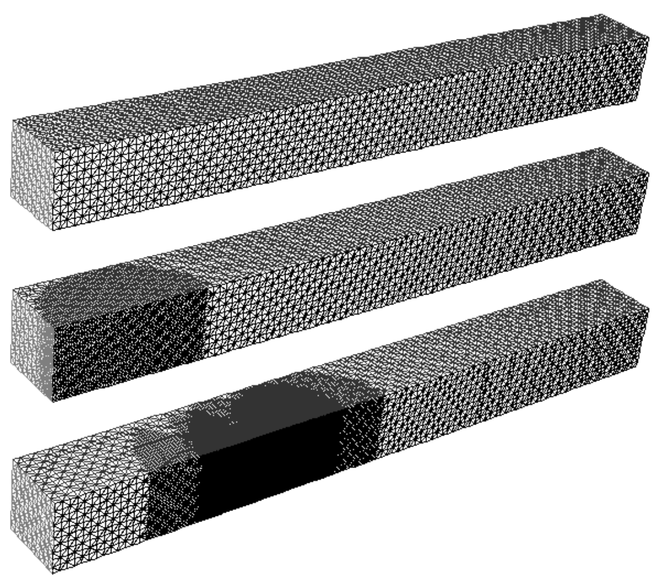


Figure 18. Mesh adaptations (initial, one, and five); simulated shock wave.

5.3 Shock wave experiment

The experiments described in section 3.4 utilize *synthetically* generated loads to demonstrate the effectiveness of the SBN-based load balancing algorithms as compared to other popular balancing algorithms. The experiments in chapter 4 show the adaptability of SBN-based load balancing by applying it to an unstructured mesh adaptation application which is not normally traditionally solved using a general purpose dynamic load balancer. In this chapter, a dynamic mesh experiment allows some additional comparisons between the SBN approach and the PLUM framework to be drawn and further demonstrates the effectiveness of the SBN dynamic balancing approach.

The experiment to be analyzed simulates the propagation of a simulated shock wave within a cylindrical volume. The environment simulated is unsteady and the adapted region is strongly time-dependent. The propagated shock wave is applied to initial grid shown at the top of figure 18. The test case is generated by refining all elements within a cylindrical volume moving left to right across the domain with constant velocity, while coarsening previously refined elements in its wake. Performance is measured at nine successive adaptation levels. The size of the computational mesh increased from 50,000 to 1,833,730 over the nine levels of adaptation.

5.3.1 Performance metrics

The following metrics are chosen to evaluate the effectiveness of the SBN-based load balancer when processing an unsteady adaptive mesh. Recall that v denotes a vertex to be processed and P is the total number of processors.

- **Maximum redistribution cost:** The goal is to capture the total cost of packing and unpacking data, separated by a barrier synchronization. Since a processor can either be sending or receiving data, the overhead of these two phases are modeled as a sum of two costs in the following metric:

$$\texttt{MaxSR} = \max_{p \in P} \Big\{ \sum_{v \text{ sent from } p} Remap_p^v \Big\} + \max_{p \in P} \Big\{ \sum_{v \text{ received by } p} Remap_p^v \Big\}.$$

Since `MaxSR` pertains to the processor that incurs the maximum redistribution cost, a reduction in the total data redistribution overhead can be guaranteed by minimizing `MaxSR`.

- **Load imbalance factor:** This metric is formulated as:

$$\texttt{LoadImb} = (\max_{p \in P} \texttt{QWgt}(p)) \;/\; \texttt{WSysLL}.$$

The `LoadImb` factor should be as close to unity as possible.

- **Cut percentage:** The runtime interaction between adjacent vertices residing on different processors is represented as:

$$\texttt{Cut\%} = 100 \times \sum_{p \in P} \sum_{v \text{ assigned to } p} Comm_p^v \Bigg/ \sum_{e \text{ in mesh}} Comm^e ,$$

where $Comm^e$ is the weight of edge e in the adaptive mesh. The `Cut%` value should be as small as possible.

5.4 SP2 experimental results

In this section, experimental results are described that provide information to analyze the overhead associated with SBN-based balancing. The SBN-based load balancing algorithm was first implemented using MPI on the wide-node IBM SP2 located at NASA Ames Research Center, and tested with data obtained from adaptive calculations of the time-dependent unsteady simulated shock wave experiment. These results are described in greater detail in section 5.6 where performance results are charted on table 11. In addition to achieving excellent load balance, the redistribution cost is significantly reduced compared to the results obtained in experiments using the PLUM framework. However, the edge cut percentages are somewhat higher. The results demonstrate that

Table 6. Communication overhead of the SBN load balancer

P	Balancing Messages(bytes)	Balancing Bandwidth(%)	Migration Messages(bytes)	Migration Bandwidth(%)
2	342,456	0.00	3,918,912	3.67
4	149,964	0.00	7,939,344	7.44
8	463,340	0.01	25,397,376	23.79
16	581,432	0.02	30,453,888	28.53
32	1,550,292	0.12	38,244,384	35.83

the SBN strategy reduces the redistribution cost at the expense of a somewhat higher communication cost.

Tables 6, 7 and figures 19, 20 show measurements of overheads due to message passing and processing according to experiments run on the SP2. Table 6 gives the number of bytes that were transferred between processors during the load balancing and the job distribution phases. The number of bytes transferred is also expressed as a percentage of the available bandwidth. A wide-node SP2 has a message bandwidth of 36 megabytes/second and a message latency of 40 micro seconds. Figure 19 plots this message passing overhead graphically, and demonstrates that the cost of vertex migration is significantly greater than the cost of actually balancing the system load. This is not unexpected. An extrapolation of the results using an exponential curve-fitting program indicates that normal speedup will not scale past 128 processors. The formula derived by the curve-fitting program for total overhead, T_{ovhd}, due to processing and message passing is obtained as:

$$T_{ovhd} = 2.4870 \times P^{\log 1.8041} + 0.1023 \times P^{\log 1.5451} + 12215.8776 \times P^{\log 0.5777}.$$

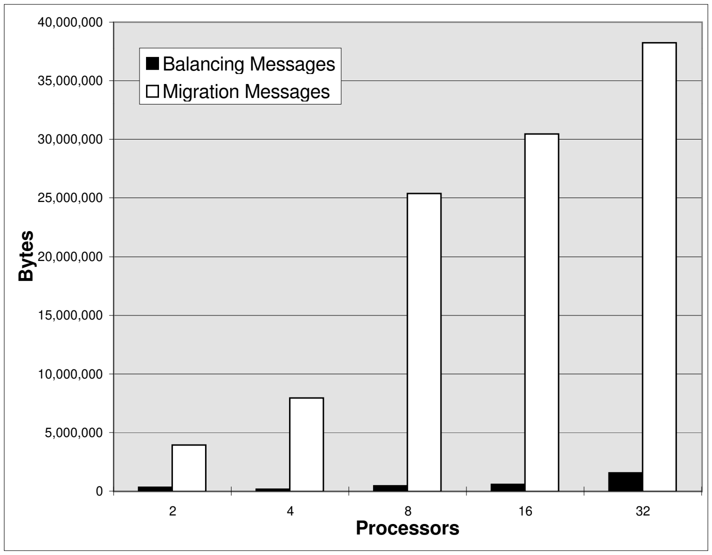


Figure 19. SBN load balancing related communication overhead.

As expected, much of the overhead is due to the latency associated with transmitting many small messages which is represented by the dominating term, $P^{\log 1.8041}$, where P is the total number of processors used. However, since this exponent is less than one, the overhead is asymptotically sub-linear.

Table 7 shows the fraction of time spent in the SBN load balancer compared to the execution time required to process the mesh adaptation application. The three columns correspond to three categories of load balancing activities: (i) the time needed to handle balance related messages, (ii) the time needed to migrate messages from one processor to another, and (iii) the time needed to select the next vertex to be processed. Figure 20 shows graphical plots. The results show that processing related to the selection of vertices is the most expensive phase of the SBN load balancer. However, the total time required to load balance is still relatively small compared to the time spent processing the mesh.

Table 7. Percentage overhead of the SBN load balancer

P	Balancing Activity	Migration Activity	Vertex Selection
2	0.0053	0.0014	0.4530
4	0.0087	0.0069	2.0381
8	0.1745	0.0569	2.8386
16	0.2669	0.0629	0.8845
32	0.7885	0.5407	2.3062

In conclusion, these experimental results complement the results of chapter 4 and again demonstrate that the proposed SBN-based dynamic load balancer is effective in processing adaptive grid applications, thus providing a global view across processors.

5.4.1 Origin 2000 Experimental results

In this subsection, the SBN methodology was directly ported from the SP2 to the Origin2000 machine without any required code modifications (Minor revisions as will be discussed, were made to refine the approach.) and additional experiments conducted. These experiments demonstrate the architecture independence of the SBN-based load balancer.

The plots in figure 21 illustrate the effect upon `Cut%`, `MaxSR`, and `LoadImb` when the experiments were run on the Origin2000 as compared with those parameters for SP2. `LoadImb` percentages are almost identical with those on SP2. `Cut%` values are consistently larger on the Origin2000. But `MaxSR` values are larger on the Origin2000 when $P < 16$.

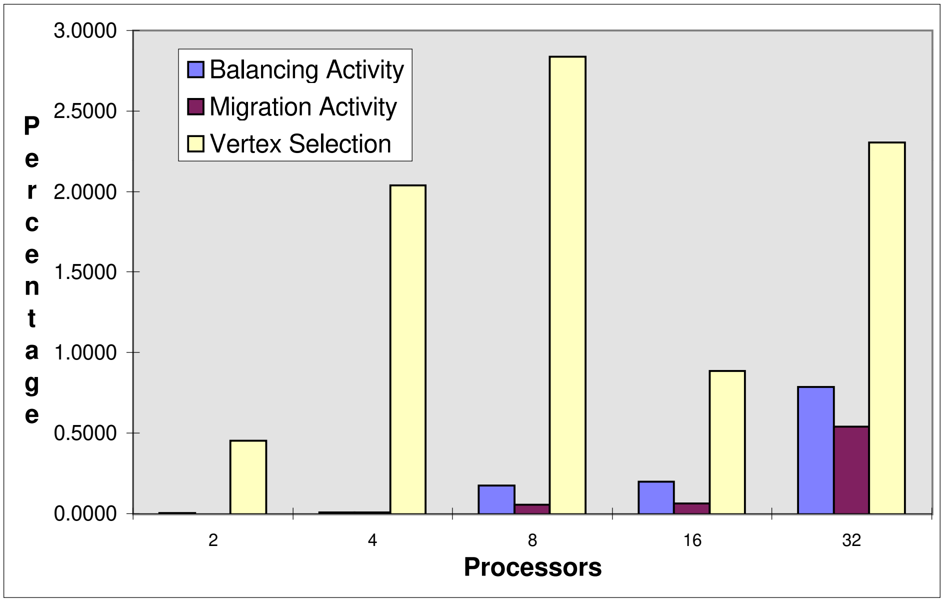


Figure 20. SBN load balancing related processing overhead.

Some of the differences in performance results on these two machines are due to additional refinements that were implemented prior to running the experiments on the Origin2000. The refinements are summarized in the following itemization:

- `QWgt`(p) was revised to include the cost of migrating the data sets that are assigned to p but whose corresponding vertices are to be processed elsewhere. For example, if vertex v is to be processed by processor q but its data set is resident in p, $Remap_q^v$, is added to `QWgt`(p). This refinement more accurately models processor load than what was reflected in the `QWgt`(p) definition presented in section 4.2. Note that this modification increases the anticipated cost of vertex migration. As a result, the SBN-based balancing algorithm tolerates a greater cut size.

- The algorithm for distributing jobs was modified somewhat for the Origin2000 experiments. The purpose of this refinement is to guarantee that the processor

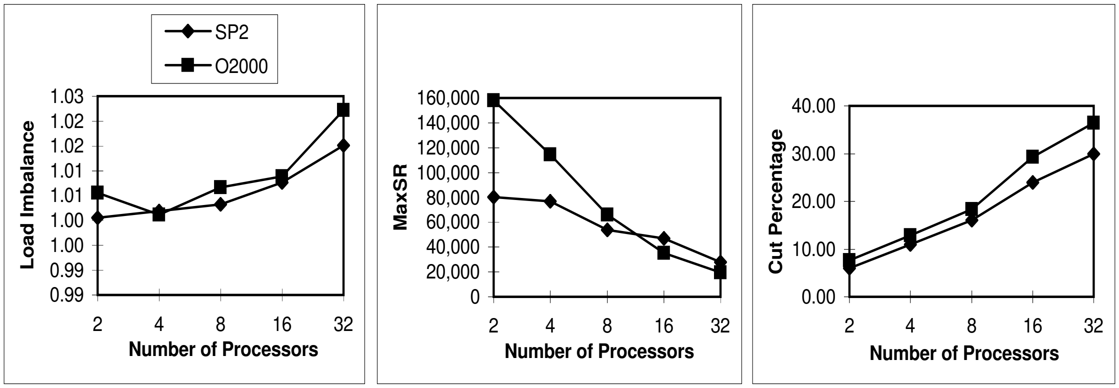


Figure 21. `LoadImb`, `MaxSR`, and `Cut%` on the SP2 and Origin2000.

initiating load balancing always gets sufficient load to process, thereby reducing the total number of balancing related messages that need to be processed (the initial processor receiving a balance request returns half of its job queue to the calling processor.) A side effect of this refinement is to increase the number of vertices migrated, especially when the value of P is small.

- Another refinement was made to migrate data sets corresponding to groups of vertices at a time. This allows for fewer total migration messages. In the experiments, the volume of migration messages were reduced by more than 80% as a result of this change.

5.5 SBN comparisons to PLUM

Both the PLUM and the SBN-based load balancers were implemented using the MPI message-passing library. Performance results, averaged over nine levels of adaptation, are presented in table 8. These values were obtained on the Origin2000, and are platform independent. Both strategies achieve excellent load balance quality (`LoadImb`;) however, the SBN-based algorithm significantly reduces the redistribution cost (`MaxSR`) from that obtained with PLUM, more so for the larger numbers of processors. Compared to PLUM, the SBN algorithm also incurs a smaller overhead in terms of the total communication volume (in MBytes,) which is proportional to the TotalV metric. (Observe that the total communication volume increases with the number of processors, while the MaxSR metric decreases.) On the other hand, the MeTiS partitioner used in PLUM generates dramatically smaller edge cuts (Cut%,) thereby minimizing the runtime interprocessor communication. Thus there is a trade-off here: SBN reduces the data redistribution overhead at the expense of greater runtime communication. This is due to the lazy approach of data migration used by the SBN-based load balancer, but may be an acceptable compromise for applications where the data migration cost is dominant.

Table 9 shows the fraction of time spent in the load balancer compared to the total time to process the mesh adaptation application for both PLUM and SBN. Notice that the partitioning overhead of PLUM is dramatically less than the balancing cost of SBN. This is due to the state-of-the-art MeTiS partitioner, which uses the dual graph information to efficiently create a new mesh distribution. The SBN balancer, on the other hand, is dominated by the cost of vertex selection. The algorithm dynamically chooses the next dual graph vertex to be processed, depending on specific runtime criteria. Thus, vertex selection is not as efficient as parallel partitioning, since the balancing data are not available a priori as in the PLUM framework. However, the cost of remapping within PLUM is significantly more expensive than SBN distribution. During PLUM remapping, useful processing is suspended while the mesh is redistributed and data objects are appropriately rebuilt. The SBN approach allows processing to continue while the load is dynamically balanced. This allows the communication overhead of distribution to be overlapped with the processing of the mesh application, substantially reducing the data movement cost compared to traditional remapping schemes. Finally, observe that the Origin2000 is responsible for a smaller overall percentage of remapping/distribution compared to the SP2, for both balancing strategies. This is due to the superior network performance of the Origin2000.

Table 8. Performance results of the PLUM and SBN-based balancers

	LoadImb		Cut%		MaxSR		Comm. Vol.	
P	PLUM	SBN	PLUM	SBN	PLUM	SBN	PLUM	SBN
2	1.00	1.01	3.29	7.65	157,542	158,085	7.625	3.921
4	1.00	1.00	4.01	12.81	143,227	114,652	14.264	7.250
8	1.01	1.01	5.77	18.33	129,859	65,824	26.092	18.356
16	1.02	1.01	7.81	29.25	103,090	35,389	36.787	27.658
32	1.04	1.02	10.94	36.47	63,270	19,446	41.113	35.268

5.5.1 Advantages of dynamic balancing

The SBN load balancing algorithms are designed to run dynamically without the need for a separate partitioning process. This is a significant advantage over the existing approaches taken by most parallel adaptive mesh applications implemented to date. Those methods must suspend processing when processor loads become imbalanced. During the suspension, mesh vertices are reassigned and the corresponding data sets are remapped. Another advantage of the SBN approach is that it allows overlapping of the computational, communication, and redistribution phases which would lead to further reductions in the overall execution time. Existing methods which separately process the computational, partitioning, and remapping phases of adaptive mesh applications cannot achieve this overlap.

Table 9. Percentage overhead of the PLUM and SBN-based balancers

	PLUM				SBN			
	Partitioning		Remapping		Balancing		Distribution	
P	SP2	O2000	SP2	O2000	SP2	O2000	SP2	O2000
2	0.13	0.14	1.45	1.11	0.86	0.75	0.04	0.03
4	0.15	0.15	2.56	2.01	1.15	0.99	0.11	0.08
8	0.17	0.18	3.52	2.83	2.11	1.48	1.48	0.11
16	0.32	0.34	4.04	3.31	2.75	1.89	1.89	0.16
32	0.45	0.50	4.40	3.67	3.01	2.52	2.52	0.20

5.6 SBN pre-partitioner

To further test the effectiveness of the SBN approach, a pre-partitioner was implemented which can optionally be run prior to each mesh adaptation phase. Since state-of-the-art parallel partitioners execute quickly (less than a second for a million vertices on 64 processors,) it would be interesting to determine whether separately running a front-end partitioner would have significant benefit on the resulting communication and/or redistribution overhead. Our pre-partitioner computes a P-way partition of the mesh vertices and then uses the default partition-to-processor assignment as the starting point for subsequent SBN-based load balancing. This pre-partitioner is non-standard in the sense that it partitions the mesh based on `QWgt`(p) values, which take into consideration all three factors of computation, communication, and redistribution. In addition, the data associated with each mesh vertex are not migrated after the partitioning process is completed. The actual data movement takes place during mesh adaptation as vertices are processed with SBN load balancing in effect.

Since the initial partition-to-processor assignment represents only a starting point, the partitioning capabilities inherent in the SBN-based balancing algorithm could significantly alter the processor assignments as the mesh is subsequently processed. These adjustments occur dynamically as individual processors become overloaded or underloaded. Our experiments demonstrate that the SBN pre-partitioner produces lower communication costs and higher data remapping costs than when the pre-partitioning option is not used (see tables 10 and 11.) Although the SBN pre-partitioner may be of limited value for those adaptive mesh applications where remapping costs dominate communication costs, its algorithm is sketched in the following paragraphs the sake of completeness.

The SBN pre-partitioner is a diffusive algorithm which uses the original mesh as a starting point and improves balancing by reassigning vertices among processors. It computes a P-way partition such that $\texttt{QWgt}(p)$ is approximately equal for all the processors and $\texttt{WSysLL}$ is minimized. Since $\texttt{QWgt}(p)$ is the sum of the computational, communication, and redistribution costs, this is a somewhat stronger requirement than that considered in some other approaches [41, 45] where the mesh is partitioned to equalize the total computational cost while minimizing the total number of cut edges. Therefore, if only a few processors incurred most of the communication overhead, significant idle time could result during processing.

The pre-partitioner differs from the partitioning capabilities inherent in the SBN load balancer in that multiple iterations are executed to find an optimal P-way partition. Here, an *iteration* is defined as a sequence of vertex reassignments from one processor to another. During an iteration, each vertex is allowed to be reassigned at most once. Reassignments are made so that vertices in processor p with $\texttt{QWgt}(p) > \texttt{WSysLL}$ are assigned to the processor q with the minimum $\texttt{QWgt}(q)$ value.

Each vertex to be reassigned is adjacent to a random subset of vertices chosen and belonging to q. First, the $\Delta\texttt{Cut}$ value is computed for all adjacent vertices assigned to processors other than q. The non-local adjacent vertex $\texttt{MinV}$, one with the smallest $\Delta\texttt{Cut}$, is then added to the set of vertices assigned to q. In addition, a breadth-first search is performed on the vertices adjacent to $\texttt{MinV}$ that are not assigned to q but to p where $\texttt{QWgt}(p) > \texttt{WSysLL}$. These vertices are also assigned to q.

At the end of an iteration when all vertices have been considered, the partitioner computes the load imbalance factor $\texttt{QWgt}(r)/\texttt{WSysLL}$, for the processor r with the largest value of $\texttt{QWgt}(r)$. If the load imbalance factor is greater than a specified threshold (1.75 in our experiments,) the Kernighan-Lin refinement procedure [47] is invoked to further reduce $\texttt{WSysLL}$.

Initially, the pre-partitioner is set to execute a fixed number of iterations, $\texttt{ItNum}$. In our experiments, $\texttt{ItNum} = 4$ produced the best results. After each iteration, the total number of iterations is increased by $\texttt{ItNum}$ if a new minimum $\texttt{WSysLL}$ is achieved. The algorithm terminates when all iterations are completed.

5.6.1 Pre-partitioner experimental results

Table 10 presents the results of processing the adaptive mesh with the SBN load balancer when running the SBN pre-partitioner between mesh adaptations. Table 11 presents similar results, but the SBN pre-partitioner is not invoked between adaptation phases. Tables 12 and 13 respectively chart the results achieved using the ParMeTiS and DMeTiS partitioners within the PLUM environment. Note that tables 12 and 13 do not show results corresponding to all values of $P = 2$, 4, 8, 16, and 32. Only those data sets that were available are included.

The following metrics are referenced in tables 10 and 11:

Pre-Part Cut%: Initial edge cut projection before running the Pre-partitioner.

Pre-Exec Cut%: Mesh edge cut immediately before processing a mesh adaptation.

Post-Exec Cut%: Actual cut realized after processing a given adaptation level.

Overall, the SBN-based approach achieves excellent load balance whether the pre-partitioner is active or not. As shown in tables 10 and 11, `LoadImb` $= 1.02$ for $P = 32$. When $P \leq 8$, an ideal load imbalance factor of 1.00 is achieved for most of the adaptation levels. In contrast, this factor is respectively 1.04 and 1.59 using ParMeTiS and DMeTiS under the PLUM environment (see tables 12 and 13.)

The `MaxSR` metric indicates the amount of redistribution cost incurred while processing the adaptive mesh. The SBN "lazy" approach to migration of vertex data sets produces significantly lower values than those achieved by ParMeTiS or DMeTiS under PLUM. For example, for $P = 32$, table 11 shows `MaxSR` $= 28,031$, which is significantly less than the corresponding value in table 12 (`MaxSR` $= 63,270$) and in table 13 (`MaxSR` $= 62,542$.) Additionally, when the SBN balancer is used with the partitioner, a higher `MaxSR` value results when $P \geq 8$ than when the partitioner is not active (see table 11.) This indicates a trade-off of lower redistribution cost for higher cut percentage. The pre-partitioner will allow an increase in the redistribution cost if it is compensated by a larger improvement in the communication cost. Comparing `Pre-Part Cut%` and `Pre-Exec Cut%` from table 10, it can also be concluded that `Cut%` degrades as the pre-partitioner executes. This result is consistent with the observations drawn from the PLUM experiments.

Table 10 shows an SBN cut percentage that is more than double compared to those reported by ParMeTiS (21.29% vs. 10.94% for $P = 32$.) This difference in the cut percentage is significantly lower when compared to the results obtained with DMeTiS (21.29% vs. 20.22%.) These results could reflect the effectiveness of the partitioners being used rather than whether the SBN balancer will always produce higher communication costs. For example, in table 10, the cut percentage often decreases during execution of the SBN balancer. This phenomenon did not occur while experimenting with ParMeTiS or DMeTiS under PLUM.

Note that the cut percentage is about 1.3 times higher when the SBN pre-partitioner is not active (see tables 10 and 11.) This implies that it may be useful to initially partition the mesh to compute a starting point for subsequent SBN load balancing when high communication cost is a critical factor.

In conclusion, these experimental results demonstrate that the proposed SBN-based dynamic load balancer has definite advantages over the traditional semi-dynamic frameworks in achieving a lower cost of data redistribution; but a higher communication cost. In many mesh applications in which cost of data redistribution dominates the cost of communication and processing, the SBN balancer would be preferred over a semi-dynamic balancing approach.

5.7 Complexity analysis

In this subsection, overhead associated with the execution of the SBN-based load balancer and/or pre-partitioner is analyzed while processing the computational mesh. Where possible, both analytic formulas and experimental data are presented.

The SBN pre-partitioner computes a P-way partition (where P is the number of available processors) of the adaptive mesh before each phase of execution. When $P = 32$, the partitioning time was measured to be about 120 seconds, which is much higher than the average processing time (less than one second) required by the ParMeTiS and DMeTiS partitioners. Note, however, that the SBN pre-partitioner runs sequentially and was not optimized for peak performance. Furthermore, the SBN pre-partitioner does not use the multilevel approach used by ParMeTiS and DMeTiS that results in significant speedup.

The overhead due to the SBN-based load balancer has four components: (i) the choice of the next vertex to be processed; (ii) the selection of the set of vertices to be

Table 10. SBN mesh adaptation results with pre-partitioning

Adaptation Level		Pre-Part Cut% Cut%	Pre-Exec Cut% Cut%	Post-Exec Cut% Cut%	MaxSR	LoadImb
$\boxed{P=2}$	1	0.09	1.76	0.95	9,606	1.00
	2	1.21	2.82	1.60	41,926	1.00
	3	0.59	3.36	2.60	178,631	1.00
	4	3.28	4.00	2.31	118,679	1.00
	5	2.94	3.02	2.39	112,437	1.00
	6	4.36	3.88	2.93	87,517	1.00
	7	2.76	2.53	1.78	75,925	1.00
	8	0.51	3.14	2.08	223,160	1.00
	9	2.55	2.87	2.18	103,772	1.00
Average		2.03	3.04	2.09	105,739	1.00
$\boxed{P=4}$	1	2.26	3.67	2.58	6,937	1.00
	2	3.37	4.11	3.13	24,382	1.00
	3	3.60	6.03	4.96	81,348	1.00
	4	5.73	5.51	4.56	85,345	1.00
	5	6.23	6.58	4.89	101,070	1.00
	6	6.25	6.29	5.12	51,018	1.00
	7	5.95	8.22	6.72	145,850	1.00
	8	7.45	8.36	6.86	92,430	1.00
	9	6.05	9.63	4.99	69,413	1.00
Average		5.21	6.49	4.87	73,088	1.00
$\boxed{P=8}$	1	6.66	7.16	6.05	6,939	1.01
	2	7.60	7.56	6.17	22,833	1.00
	3	7.85	8.48	7.33	90,132	1.02
	4	7.78	17.35	14.67	139,439	1.00
	5	12.64	12.19	11.58	138,671	1.00
	6	7.97	11.19	9.88	123,433	1.01
	7	12.09	11.81	10.74	130,199	1.01
	8	12.39	10.99	9.93	123,223	1.00
	9	7.93	10.06	8.90	158,867	1.01
Average		9.21	10.75	9.47	103,748	1.01
$\boxed{P=16}$	1	15.36	11.48	11.01	5,647	1.01
	2	13.15	11.71	11.37	26,263	1.01
	3	12.89	13.02	12.59	107,173	1.01
	4	8.38	22.60	21.39	209,028	1.01
	5	17.08	14.05	14.28	122,902	1.01
	6	14.66	13.14	12.55	100,962	1.01
	7	13.08	21.37	19.12	135,900	1.01
	8	16.63	13.57	14.76	126,161	1.01
	9	12.13	23.36	21.57	102,203	1.01
Average		13.71	16.03	15.41	104,027	1.01
$\boxed{P=32}$	1	21.59	13.57	15.50	3,764	1.03
	2	18.20	14.49	14.28	10,784	1.02
	3	14.59	20.25	19.78	53,423	1.02
	4	13.43	21.14	23.35	154,009	1.01
	5	15.95	28.07	29.09	196,821	1.01
	6	19.94	21.65	22.19	117,254	1.02
	7	17.07	23.23	23.82	90,404	1.02
	8	18.44	17.62	20.30	90,322	1.01
	9	14.58	23.74	23.31	116,354	1.01
Average		17.09	20.42	21.29	92,571	1.02

Table 11. SBN mesh adaptation results (no pre-partitioning)

Adaptation Level		Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$P=2$	1	0.09	4.64	6,974	1.00
	2	3.14	6.18	30,538	1.00
	3	5.36	6.08	57,724	1.00
	4	3.93	3.86	20,646	1.00
	5	2.91	5.32	76,893	1.00
	6	2.33	4.62	103,544	1.00
	7	2.23	5.86	140,904	1.00
	8	2.83	6.14	153,735	1.00
	9	3.10	6.89	129,374	1.00
	Average	2.88	5.51	80,037	1.00
$P=4$	1	2.26	8.15	4,078	1.01
	2	7.22	10.01	26,187	1.00
	3	9.44	11.69	64,110	1.00
	4	9.16	9.48	46,406	1.00
	5	6.60	11.86	149,042	1.00
	6	9.83	10.89	94,269	1.00
	7	6.58	8.00	50,337	1.00
	8	2.79	15.31	170,408	1.00
	9	11.53	11.48	85,152	1.00
	Average	7.27	10.76	76,665	1.00
$P=8$	1	6.66	10.77	2,518	1.01
	2	13.93	14.98	11,109	1.00
	3	15.11	18.16	46,088	1.00
	4	14.65	15.83	53,032	1.00
	5	11.09	16.48	69,583	1.00
	6	11.02	15.91	85,982	1.00
	7	13.75	18.13	105,946	1.00
	8	12.84	19.51	28,974	1.00
	9	15.34	17.35	80,477	1.00
	Average	12.71	16.35	53,745	1.00
$P=16$	1	15.36	20.61	1,767	1.01
	2	24.82	25.56	7,259	1.00
	3	24.40	27.45	36,031	1.01
	4	20.60	22.77	43,943	1.01
	5	16.11	24.27	71,736	1.01
	6	17.83	22.28	66,211	1.01
	7	19.75	25.00	55,361	1.01
	8	17.83	25.30	64,796	1.01
	9	17.87	21.59	74,316	1.01
	Average	19.40	23.87	46,824	1.01
$P=32$	1	21.59	26.74	1,184	1.01
	2	30.35	32.32	4,387	1.02
	3	30.06	34.04	8,445	1.02
	4	27.28	31.43	41,783	1.01
	5	21.35	29.40	42,843	1.01
	6	24.04	29.42	42,688	1.01
	7	22.35	30.45	41,347	1.02
	8	20.59	30.48	37,006	1.02
	9	22.19	29.43	32,594	1.02
	Average	24.42	30.41	28,031	1.02

Table 12. Mesh adaptation results using ParMeTiS with PLUM

Adaptation Level	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$\boxed{P = 16}$				
1	3.16	4.38	10,088	1.02
2	5.34	7.20	25,875	1.02
3	7.27	9.71	58,887	1.03
4	5.24	8.62	134,808	1.03
5	5.77	8.17	153,154	1.04
6	4.70	8.06	122,151	1.02
7	4.47	8.45	159,037	1.02
8	5.31	7.97	132,987	1.01
9	4.18	7.75	130,824	1.01
Average	5.05	7.81	103,090	1.02
$\boxed{P = 32}$				
1	4.78	6.45	5,097	1.01
2	7.56	10.05	16,758	1.02
3	10.28	13.13	39,565	1.05
4	8.14	11.60	73,074	1.06
5	7.59	11.13	92,581	1.05
6	6.51	11.60	82,751	1.06
7	6.66	11.43	88,642	1.03
8	6.88	11.39	91,301	1.05
9	6.19	11.66	79,662	1.04
Average	7.18	10.94	63,270	1.04

migrated; (iii) the processing required to determine if load balancing is necessary; and (iv) the load balancing (communication) messages between processors. These components are analyzed in the following paragraphs.

The next vertex v is selected using a priority min-queue. Let V_p be the set of vertices to be processed at a given processor p and let E_p be the set of all internal and border edges that are adjacent to the vertices of V_p. Heap operations like create and insert/delete-min require $O(V_p)$ and $O(\log V_p)$ time, respectively. The (non-standard) remove operation can be implemented in $O(\log V_p)$ time provided a direct pointer to the entry to be removed is also maintained. For SBN processing, however, $(Wgt^v + Comm_p^v + Remap_p^v)$ must be

Table 13. Mesh adaptation results with DMeTiS and PLUM

Adaptation Level	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$\boxed{P = 32}$				
1	4.65	15.70	5,047	1.88
2	19.26	20.50	17,393	2.12
3	21.14	25.26	44,413	2.12
4	17.13	28.21	99,232	1.87
5	29.08	26.46	97,280	1.68
6	25.31	24.38	86,204	1.41
7	20.55	14.17	78,312	1.11
8	10.04	13.08	72,474	1.05
9	9.41	14.18	62,522	1.05
Average	17.40	20.22	62,542	1.59

computed so that the value of `QWgt`(p) can be maintained and $(Comm_p^v + Remap_p^v)/Wgt^v$ is needed to correctly control the ordering of the SBN priority min-queue. Each of these calculations require $O(\delta_v)$ time, where δ_v is the degree of vertex v. Therefore, the SBN priority queue (heap) creation requires $O(V_p + \sum_{v \in V_p} \delta_v) = O(V_p + E_p)$. Similarly, each heap insertion, delete-min, and remove operation completes in $O(\log V_p + \delta_v)$.

The vertex migration in SBN involves first selecting a random set, R, of vertices from the mesh. The vertex, $m \in R$, with the smallest Δ`Cut` value is chosen for migration. Each Δ`Cut` calculation completes in $O(\delta_r)$ where $r \in R$. Therefore, the time required to select the initial vertex for migration is

$$O(\sum_{r \in R} \delta_r) \approx O(|R| \times \delta_{avg}) \text{ where } \delta_{avg} \text{ is the average degree of a vertex in the mesh.}$$

Next, the mesh is searched in a breadth-first manner to choose an additional set, V_{mig}, of vertices for migration. In the experiments that were reported, $|V_{mig}|$ averages less than ten to satisfy a load balancing request. Furthermore, a single breadth-first search almost always finds enough vertices to migrate. The time required to complete the breadth-first search is approximated by

$$O(|V_{mig}| + \sum_{v \in V_{mig}} \delta_v) \approx O(|V_{mig}| + |V_{mig}| \times \delta_{avg}).$$

Finally, each vertex to be migrated must be removed from the SBN priority queue. This operation is needed so that the vertices migrated will no longer be considered for local processing. Since $|V_{mig}| + 1$ removal operations from the heap are required, the complexity for this step is

$$O((|V_{mig}| + 1) \times \log V_p \ + \sum_{v \in V_{mig} \cup m} \delta_v) \approx O(|V_{mig}| \times (\log V_p + \delta_{avg})).$$

Combining the terms just derived and considering that $|R|$ is a constant, the overall asymptotic time complexity for job migration is given by

$$T_{mig} = O(|V_{mig}| \times (\log V_p + \delta_{avg})).$$

Each processor must periodically check whether a load balancing operation is to be initiated or if load balancing messages from other processors are to be processed. If processors check too frequently, the associated overhead could be too high. On the other hand, infrequent checks for load balancing activity could lead to excessive idle time. The following analysis can be used to minimize this overhead without significantly increasing processor idle time.

If f is the frequency of a processor checking for load balancing activity, we can expect the response time to process a load balance message to be $2/f$ on average. Every time the SBN load balancer is invoked, balancing and distribution messages pass through $3 \log P$ stages of communication. Therefore, the total response time to balance the system load is $(6 \log P)/f$. If J_{avg} is the average number of jobs processed per unit time, the threshold `MinTh` should be set to a value such that load balancing will be triggered when $\texttt{QWgt}(p) < \lceil 6 \log P \times J_{avg}/f \rceil$ to avoid excessive idle time. In other words,

$$\texttt{MinTh} \geq \left\lceil \frac{6 \log P \times J_{avg}}{f} \right\rceil.$$

As an example, let $P = 32$. If an application checks for load balancing activity 60 times per second (i.e., $f = 60$) and $J_{avg} = 1000$ jobs are processed per second, then MinTh ≥ 500.

CHAPTER 6

A LATENCY-TOLERANT IPG PARTITIONER

NASAs Information Power Grid (IPG) is an infrastructure designed to harness the power of geographically distributed computers, databases, and human expertise, in order to solve large-scale realistic computational problems. This type of a metacomputing environment is necessary to present a unified virtual machine to application developers that hides the intricacies of a highly heterogeneous environment and yet maintains adequate security. For the IPG to be effectively utilized, general purpose utilities are needed that are designed to execute in low-bandwidth highly distributed configurations. This chapter describes a latency tolerant partitioner, called MinEX, that is specifically designed for a distributed environment such as the IPG. MinEX not only balances processor workloads but minimizes data movement and runtime communication for applications such as adaptive meshes executing in a parallel distributed fashion on the IPG. Although the experimental results presented in this document are conducted using adaptive meshes, MinEX is a general purpose partitioner and can easily be utilized for a wide variety of applications.

6.1 Introduction

Many applications can benefit from a distributed infrastructure such as the Information Power Grid (IPG). The I-Way experiment [22], for example has identified several classes of applications that would immediately be applicable. Examples include:

- Desktop coupling to remote supercomputers to provide access to large databases and high-end graphics facilities.

- User access to sophisticated instruments through remote supercomputer connections utilizing virtual reality techniques [21].
- Remote interactions with supercomputer simulations [23, 24].

With each order of magnitude of bandwidth improvement, it is anticipated that many other applications can benefit from computing grid architectures. Therefore, the research community has actively explored various approaches as to how computational grids should be constructed. A comprehensive survey of current technology can be found in [31]. The Condor system [55] for example, developed to manage research studies at workstations around the world, is an example of an early success. However, Condor did not adequately deal with security issues that are important for a general computational grid implementation. Other grid-based systems that were developed include Nimrod [1], NetSolve [10], NEOS [14], Legion [39], and CAVERN [53]. Recently the Globus Project [38] has successfully developed an infrastructure toolkit [32] of general purpose utilities that are modular and can be utilized to present a transparent virtual machine environment to the user. Mechanisms exist within Globus to share remote resources, provide adequate security, and allow MPI-based message passing. Due to its general, portable, and modular nature, Globus was chosen by NASA as the middleware to implement the IPG.

The goal of the MinEX partitioner is different from that of most partitioners: Instead of attempting to balance the processing load, the objective is to minimize the total runtime of the application. This approach counters the possibility that perfectly balanced loads can still incur excessive communication and redistribution costs while the application is processed.

This chapter is organized as follows. Section 6.2 defines metrics that will be used in this and in chapter 7. Section 6.3 gives an overview of the MeTiS partitioner to which

MinEX will be compared in experiments that are presented. Section 6.4 describes the MinEX partitioner.

6.2 Definitions

User programs pass the MinEx partitioner a graph as input so that the vertices of this graph can be assigned among the processors of the grid. Each vertex v of the supplied graph to has two weights, $\texttt{PWgt}_v$ and $\texttt{RWgt}_v$, while each edge (v, w) has two weights, $\texttt{CWgt}_{(v,w)}$ and $\texttt{CWgt}_{(w,v)}$. These weights refer respectively to the processing, data remapping, and communication costs associated with processing a graph vertex. Note that edges have two weights in case the communication cost between (v, w) and (w, v) are not equal (e.g., directed graphs.)

To realistically predict performance on a variety of distributed architectures, a grid configuration graph is also utilized by our new MinEX partitioner. In addition, a processor to cluster mapping `PMap[p]` is defined to determine the cluster associated with each processor p in the grid.

The configuration graph allows a distributed network of multi-computer clusters to be defined. All processors in a cluster are homogeneous and there is a constant bandwidth for intra-cluster communication. $\texttt{Proc}_c \geq 1$ represents the processing slowdown factor for a cluster relative to the others. Likewise, edges $\texttt{Connect}_{(c,d)} \geq 1$ represent the interconnect slowdown factor when a processor in cluster c communicates with a processor in cluster d. If $c = d$, $\texttt{Connect}_{(c,c)}$ represents the slowdown associated with communication between processors in the same cluster c. Note that if $\texttt{Proc}_c$ or $\texttt{Connect}_{(c,d)}$ is unity, there is no slowdown (this represents the most efficient connection in the network).

The following metrics respectively reflect the number of time units required for computation, data remapping, and communication. Note that these metrics account for the processor assignment of vertices. This is especially important in a computational grid where communication bandwidths between nodes are likely to be much smaller than on a single multiprocessor machine.

- **Processing Weight** ($\texttt{Wgt}^v$) is the computational cost to process vertex v assigned to a processor in cluster c:

$$\texttt{Wgt}^v = \texttt{PWgt}_v \times \texttt{Proc}_c.$$

- **Communication Cost** ($\texttt{Comm}_p^v$) is the cost to interact with all vertices adjacent to v but whose data sets are not local to processor p (assuming that v is assigned to p.) If vertex w is adjacent to v and c and d are the clusters respectively associated with the processors assigned to v and w, $\texttt{Comm}_p^v$ is computed as follows:

$$\texttt{Comm}_p^v = \sum_{w \notin p} \texttt{CWgt}_{(v,w)} \times \texttt{Connect}_{(c,d)}).$$

- **Redistribution Cost** ($\texttt{Remap}_p^v$) is the overhead to copy the data set associated with v to another processor from p. Note that the redistribution cost incurred at p includes the operations of packing data and initiating transmission. The redistribution cost incurred by the processor receiving v is the sum of the communication time and the cost of unpacking and merging the data into existing data structures:

$$\texttt{Remap}_p^v = \begin{cases} \texttt{RWgt}_v \times \texttt{Connect}_{(c,d)} & \text{if } \; c \neq d \\ 0 & \text{if } \; c = d \end{cases}.$$

Assume that c is the cluster to which vertex v is assigned, and d is the cluster associated with the processor to which v is to be relocated. If the data set for v is already assigned to p, no redistribution cost is incurred (i.e., $\texttt{Remap}_p^v = 0$.) Similarly, if the data sets of all the vertices adjacent to v are also assigned to p, the communication cost, $\texttt{Comm}_p^v$, is 0.

- $\texttt{CData}_p^v$ denotes the cost to processor p for packing and unpacking data relating to inter-processor communication. If vertex v is assigned to processor p, processor p is associated with cluster c, and w is a vertex adjacent to v; $\texttt{CData}_p^v$ is computed as:

$$\texttt{CData}_p^v = (\texttt{CWgt}_{(v,w)} + \texttt{CWgt}_{(w,v)}) \times \texttt{Proc}_c.$$

The first term within the parenthesis indicates processor p's cost to pack communication data in preparation for transmission. The second term indicates the cost to unpack communication data after receiving transmission.

- $\texttt{RData}_p^v$ denotes the cost to processor p for packing and unpacking data sets for vertex reassignment. If vertex v is assigned to processor p and processor p is associated with cluster c, $\texttt{RData}_p^v$ is computed as:

$$\texttt{RData}_p^v = (\texttt{RWgt}^v_{(p,q)} + \texttt{RWgt}^v_{(q,p)}) \times \texttt{Proc}_c.$$

The first term within the parenthesis indicates the cost to pack data sets to be reassigned from processor p to q; the second term indicates the cost to unpack data sets reassigned to processor p from q.

Additional metrics used in this work are now defined:

- **Weighted Queue Length** ($\texttt{QWgt}(p)$) is the total cost to process vertices assigned to p:

$$\texttt{QWgt}(p) = \sum_{v \text{ assigned to } p} (\texttt{Wgt}^v + \texttt{Comm}_p^v + \texttt{Remap}_p^v + \texttt{CData}_p^v + \texttt{RData}_p^v).$$

- **Total System Load** ($\texttt{QWgtTOT}$) is the sum, over all processors, of $\texttt{QWgt}(p)$.

- **Heaviest Load** (`MaxQWgt`) is the maximum value of `QWgt`(p) over all processors, and indicates the total time required to process the application.
- **Lightest Load** (`MinQWgt`) is the minimum value of `QWgt`(p) over all processors, and indicates the workload of the most lightly-loaded processor.
- **Average Load** (`WSysLL`) is `QWgtTOT`/P, where P is the total number of processors.
- **Load Imbalance Factor** (`LoadImb`) represents the quality of the partitioning and is `MaxQWgt`/`WSysLL`.

6.3 Overview of MeTiS

MeTiS is a multi-level state-of-the-art general purpose partitioner. It functions by first coarsening a graph so that the resulting graph has half the number of vertices as the original. The coarsening process continues until the graph is small enough to partition efficiently. After the partitioning is completed, the refinement phase commences to restore the graph to its original size. Similar to the coarsening process, each refinement step doubles the number of vertices in the graph. Following each refinement, border vertices are also considered for moving between partitions utilizing the Kernighan/Lin algorithm [47]. A border vertex is considered for moving if load balance can be improved and if the `Cut%` remains below a user-supplied constraint.

In some ways MinEX resembles ParMeTiS. For example, both partitioners function to contract the graph to be partitioned, partition the contracted graph and then refine the graph to its original size. However, MinEX introduces several novel concepts that significantly differ from the MeTiS approach. First, the goal is to minimize runtime and not to increase load balance. In a distributed environment it is possible to have perfect balance and incur a large cost to move vertices between processors before the next adaptation can begin. The second concept introduced by MinEX is to contract and refine the graph incrementally or one vertex at a time. This allows increased flexibility in that border vertices can be reassigned at any point. The result should be less reassignments since they will be done earlier when the graph is more coarse. Finally, MinEX considers data set redistribution and latency tolerance in its partitioning logic. These considerations are particularly important in a distributed environment such as the IPG.

6.4 The MinEX partitioner

MinEX can be classified as a diffusive multilevel partitioner. The multi-level approach, originally introduced in [41], partitions a graph in three steps: contraction, partition, and refinement. Diffusive algorithms [15] utilize an existing partition as a starting point instead of partitioning from scratch. MinEX is unique in that it redefines the partitioning goal to minimizing `MaxQWgt` rather than balancing processing cost among partitions.

6.4.1 General design

Similar to other multilevel partitioners, the first step in MinEX is to contract the graph to a reasonable size. Instead of repeatedly contracting the graph in halves as is common with other multilevel partitioners, MinEX sequentially collapses one edge at a time. The advantage of this approach is that a decision can be made each time a vertex is later refined as to whether it should be assigned to another processor, making the algorithm more flexible. If $|V|$ is the number of vertices in the graph, contraction requires $O(|V|)$ steps which is asymptotically equal to the complexity of repeatedly contracting the graph in halves.

Once the graph is sufficiently contracted, the remaining vertices are reassigned according to the criteria followed by the partitioning algorithm (described in section 6.4.2.)

The graph is expanded back to its original size through a refinement process. As each vertex is refined, a decision is made as to whether it should be reassigned. This decision employs the same criteria that is followed by the partitioning algorithm. Each coarse vertex reassignment in effect reassigns all of the vertices the coarse vertex represents.

6.4.2 Partitioning criteria

To describe the criteria for deciding whether a vertex should be reassigned from one processor to another, two additional metrics, `Gain` and `MinVar`, need to be defined:

- `Gain` represents the change in `QWgtTOT` that would result from a proposed vertex move. A negative value would indicate that less processing is required after such a move. The partitioning algorithm favors vertex moves with negative or small `Gain` values that reduce or minimize overall system load.

- `MinVar` is computed using the workload ($\texttt{QWgt}(p)$) for each processor p and the average processing load (`WSysLL`) in accordance with the following formula:

$$\texttt{MinVar} = \sum_p (\texttt{QWgt}(p) - \texttt{WSysLL})^2.$$

  In other words, `MinVar` computes the variance in processor workloads from the average. The objective is to initiate vertex moves that lower this value. Since processors with large $\texttt{QWgt}(p)$ values will have large `MinVar` components, this criterion will tend to move vertices away from processors that have high runtime requirements. $\Delta\texttt{MinVar}$ is the change in `MinVar` after moving a vertex from one processor to another. A negative value indicates that the `MinVar` value has been reduced.

The partitioning decisions are made as follows. For each vertex v, the algorithm considers all edges to adjacent vertices that are assigned to other processors and computes the `Gain`, `MinVar`, `MinQWgt`, and `MaxQWgt` values that would result from moving v to each of the adjacent processors. For each possible vertex move, a user-defined function is called with these values and with the count of vertices assigned to the source processor ($\texttt{QLen}(p)$.) The user-defined function then determines whether prospective vertex moves are acceptable. For vertex moves that are deemed acceptable, the user-defined function is again called to compare prospective moves and determine which one is the best.

A typical strategy is to move the vertex with the smallest Gain where $\Delta\texttt{MinVar} < 0$. If $\texttt{Gain} > 0$, $(\texttt{Gain})^2/\Delta\texttt{MinVar} < \texttt{ThroTTle}$ where ThroTTle is a user specified parameter. Conceptually, ThroTTle acts as a gate that limits increases in Gain based upon how much of an improvement in MinVar can be achieved. A low value of ThroTTle could prevent MinEX from finding a balanced partitioning allocation; while a high value could converge at a point where runtime is unacceptable.

Table 14. Expected MaxQWgt with ThroTTle varied

	ThroTTle values						
Clusters	0	3	16	32	64	128	∞
1	1993	348	291	300	306	312	324
2	1847	748	320	304	305	318	345
3	2035	674	375	331	324	326	382
4	1868	761	412	352	328	371	425
5	1834	835	438	373	359	343	400
6	2081	898	481	391	357	361	427
7	1884	1032	505	383	371	369	414
8	1944	1102	531	434	376	380	435

Runtimes are presented in thousands of units.

Tables 14 and 15 respectively indicate how varying ThroTTle affect the expected application runtimes (MaxQWgt) and load balance quality (LoadImb). The MaxQWgt entries

Table 15. Expected LoadImb with ThroTTle values varied

	ThroTTle values						
Clusters	0	3	16	32	64	128	∞
1	7.05	1.23	1.01	1.00	1.00	1.00	1.00
2	8.54	2.74	1.26	1.14	1.04	1.00	1.00
3	7.15	2.50	1.41	1.19	1.05	1.02	1.01
4	6.63	2.82	1.58	1.26	1.07	1.03	1.01
5	6.53	3.06	1.66	1.30	1.11	1.02	1.01
6	7.31	3.25	1.81	1.40	1.08	1.02	1.01
7	6.68	3.74	1.84	1.33	1.10	1.03	1.00
8	6.90	3.92	1.94	1.43	1.13	1.06	1.00

are non-dimensioned values in thousands. These results were obtained experimentally using an adaptive mesh as a test case and illustrate how altering ThroTTle values affect MaxQWgt. Tables 14 and 15 assume a network of 32 homogeneous processors distributed over one to eight IPG nodes. The inter-cluster interconnect slowdown is assumed to be a third of the intra-cluster interconnect's bandwidth. Results show that a ThroTTle value of sixty four produces the lowest overall MaxQWgt, and that larger ThroTTle values improve LoadImb. Experiments with other network sizes using this same graph have shown that ThroTTle generally converges at values between P and $2P$. Note also that for large values of ThroTTle, better LoadImb does not necessarily imply lower MaxQWgt.

To increase efficiency, a min-heap is used with vertex pointers to heap locations to rapidly find the best move and directly remove entries without searching. The user-defined function computes the heap-key so that vertex moves can be processed in a desired order. Normally, vertex moves that lower runtime should be processed first.

6.4.3 Partitioning data structures

Described here are the data structures used by the MinEX partitioner to perform its multilevel algorithm.

`Graph` The Graph whose format is $\{|V|, |E|, \texttt{vTot}, *\texttt{VMaP}, *\texttt{VList}, *\texttt{EList}\}$ where $|V|$ is the number of active vertices, $|E|$ is the number of edges, `vTot` is the total vertex count (including merged vertices,) $*$`VMaP` is a pointer to the list of active vertices, $*$`VList` is a pointer to the complete list of vertices, and $*$`EList` is a pointer to the list of edges.

`VmaP` The list of active vertices (those that were not compressed through multilevel contraction.)

`VList` The complete list of vertices. Each vertex, v, is defined by a `VList` record as $\{\texttt{PWgt}_v, \texttt{RWgt}_v, |e|, *e, mrg, lkup, *\texttt{VMaP}, *\texttt{Heap}, bdr\}$ where $\texttt{PWgt}_v$ is the computational cost to process v, $\texttt{RWgt}_v$ is the redistribution cost to copy the data associated with v, $|e|$ is the number of adjacent edges associated with v, $*e$ is a pointer to the first edge associated with v (subsequent edges are stored contiguously,) mrg is the vertex that was merged with v during a contraction operation (-1 if not merged,) $lkup$ is the active vertex that contains v after a series of contractions (-1 if not merged,) $*vmap$ is a pointer to the position of v in the active vertex table, $*heap$ is a pointer to the heap entry that relates to v and represents a potential reassignment of v, and bdr is a boolean flag indicating whether v is adjacent to vertices assigned to other processors.

`EList` The list of edges in the graph. Each vertex v in `VList` points to its first edge in `EList` using $*e$. Each edge record is defined as $\{w, \texttt{CWgt}_{(v,w)}, \texttt{CWgt}_{(w,v)}\}$ where w is an adjacent vertex to v, $\texttt{CWgt}_{(v,w)}$ is the communication weight associated with this edge, and $\texttt{CWgt}_{(w,v)}$ is the communication weight associated with the edge (w, v). $\texttt{CWgt}_{(w,v)}$ (e.g., directed graphs). is necessary should $\texttt{CWgt}_{(v,w)} \neq \texttt{CWgt}_{(w,v)}$.

`Heap` The heap of potential vertex reassignments. Each heap record is defined as $\{\texttt{Gain}, \Delta\texttt{MinVar}, v, p\}$ which specifies the `Gain` and Δ`MinVar` that would result from reassigning v to processor p. The min-heap is normally keyed by the `MaxQWgt` value resulting from a vertex reassignment but the key is determined by a user-defined function.

`Stack` The stack of collapsed edges, (v_1, v_2) The pushed edges are refined in an order reversed from the one that they were compressed.

6.4.4 The contraction step

The partitioner uses a contraction heap to order pairs of edges (v, w) assigned to the same processor. This heap controls the order that vertices are merged and edges collapsed. Typically, the heap is keyed by negative $\texttt{CWgt}_{(v,w)}$ values. The idea is to find edges with large communication weights. Therefore, a contracted graph with a small edge cut will formed. The user can override the heap order though a user-defined function. MinEX passes, $\texttt{RWgt}_v$, $\texttt{PWgt}_v$, and $\texttt{CWgt}_{(v,w)}$ values to this function relative to prospective vertex pairs. Additionally, MinEX passes a count of the number of vertices from the original graph that are represented by vertices v and w.

To contract a pair of vertices, a merged vertex M is created and the edge (v, w) is collapsed. The edges incident on M are created by utilizing the edge lists of vertices v and w. `VMaP` is adjusted to contain the newly created vertex M and to remove v and w, $|V|$ is decremented, `vTot` is incremented, $|E|$ is increased by the number of edges created for M, and the pair (v, w) is pushed onto `Stack`.

6.4.5 Union/Find algorithm

The contraction is implemented using a set Union/Find algorithm so that edges of existing vertices remain unchanged. For example, if an existing vertex is adjacent to v, accesses to its `EList` will check whether v has been merged. If it has, $lkup$ will quickly find the appropriate merged vertex. If $lkup$ is not current ($lkup >$ `vTot`,) the Union/Find algorithm will search the chain of vertices beginning with mrg to update $lkup$, so subsequent lookups can be done efficiently. Pseudo code describing the Union/Find algorithm is given in figure 22.

```
procedure Find (v)
if (mrg == -1) return (v)
if (lkup != -1) and (lkup <= vTot)
    then return (lkup = Find (lkup))
    else return (lkup = Find (mrg))
```

Figure 22. Union/Find algorithm pseudo code.

6.4.6 The partition step

The partitioning is performed when the graph contraction process is complete. Partitioning is efficient because the number of vertices is greatly reduced. The algorithm considers every remaining graph vertex to find potential reassignments that will reduce `Gain` and `MinVar` as described in section 6.4.2. All potential vertex reassignments are added to the min-heap, and executed in heap order. After each reassignment, the heap is adjusted to reflect the new partition.

Note that the user-defined function is utilized to control the partitioning process. It has enumerated blocks of code that determine whether a potential vertex reassignment should be rejected, whether one reassignment is superior to another, and to build the key that determines the heap order. MinEX passes the changes that would occur to `MaxQWgt`, `MinQWgt`, `Gain`, `MinVar`, and processor queue length (`QLen`(p) that would result from a vertex move. These metrics allow the application to have control over the partitioning process.

6.4.7 The refinement step

The graph is restored to its original size by expanding pairs of vertices in reverse order from which they were merged. The `Stack` data structure controls the order. As pairs of vertices (v, w) are refined, merged edges and vertices are deallocated. The mrg and $lkup$ numbers are also adjusted in the vertex table. The `VMaP` table is adjusted to delete the merged vertex M and to add v and w, $|V|$ is incremented and `vTot` is decremented, and $|E|$ is decreased by the number of edges created for M. After each refinement, a check is performed to determine whether the partitioning can be improved by reassigning v or w. When reassignments are made, adjacent border vertices are also considered.

6.4.8 Latency tolerance

MinEX is able to anticipate the latency tolerance of solvers through a user-defined function that it calls during the partitioning process. For example, the following processing steps illustrate how an application can be programmed to utilize latency tolerance techniques to reduce or eliminate communication and data redistribution costs:

1. Initiate send of all data sets to be redistributed.
2. Initiate send of communication data needed by adjacent processors.
3. Process vertices that are not waiting for incoming transmissions.
4. Receive and unpack any remapped data sets destined for this processor.
5. Receive and unpack communication data destined for this processor.

6. Repeat steps 2 through 5 until all vertices are processed.

This logic proposes a strategy where processors distribute data sets and communication data as early as possible. Servicing of internal graph vertices can then take place while waiting for expected incoming messages. As data sets and communication data are received, additional communications can be initiated and vertices processed.

The user-defined function that MinEX calls contains arguments that are passed that can be manipulated to accurately model solver operation for any processor p. These arguments follow:

$\mathtt{Pproc}_p$ is the total processing weight.

$\mathtt{Cunpack}_p$ is the processing cost to unpack communication data after transmission.

$\mathtt{Cpack}_p$ is the processing cost to pack communication data before transmission.

$\mathtt{Crcv}_p$ is the total transmission cost associated with data communication.

$\mathtt{Runpack}_p$ is the processing cost to unpack data sets received after transmission.

$\mathtt{Rpack}_p$ is the processing cost of packing data sets in preparation for transmission.

$\mathtt{Rrcv}_p$ is the total transmission cost associated with data set reassignment.

$\mathtt{QLen}(p)$ is the number of vertices assigned to p.

$\mathtt{QSend}(p)$ is the number of vertices that will be relocated from p.

$\mathtt{QOwn}(p)$ is the number of vertices originally assigned to p before partitioning.

The user-defined function utilizes these totals to compute the projected $\mathtt{QWgt}(p)$ value for a particular processor. A detailed example as to how these metrics are used is given in chapter 7. It is this feature that allows MinEX to anticipate the actual solver performance as it assigns vertices.

CHAPTER 7

MINEX EXPERIMENTAL RESULTS

Limited studies were performed to determine the viability of parallel distributed computing on the IPG. In one such study [4], latency tolerance and load balancing modifications were implemented in connection with a computational fluid dynamics problem to compensate for slower communication bandwidth. Results showed that the application actually ran faster under Globus on two IPG nodes of four processors each than on a single tightly-coupled machine of eight processors. However, this result is clouded in that asynchronous message passing was supported over the high-bandwidth link but not within the single platform. In this chapter the unsteady adaptive mesh application that was introduced in section 5.3 is utilized for experiments. A simulated runtime environment is used to predict performance on a variety of wide area networks. The number of IPG nodes, the number of processors per node, and the interconnect slow downs are parameterized so that general conclusions can be drawn as to when the IPG would be suitable to solve applications of this nature.

In chapter 5, two different load balancing strategies were investigated with this application as the test case. The first strategy, called PLUM [60], is an architecture-independent semi-dynamic framework geared towards adaptive numerical solutions. The second approach uses a general-purpose topology-independent dynamic load balancer utilizing Symmetric Broadcast Networks (SBN) that was described in chapter 3. Results reported in chapter 5 indicate that both PLUM and the SBN approach have their relative merits, and that they achieve excellent load balance with minimal extra overhead.

Previous studies with this mesh application under the PLUM framework utilized a variety of general partitioners such as ParMeTiS [44], UAMeTiS [65], DAMeTiS [65], Jostle-MS [77], and Jostle-MD [77]. UAMeTiS, DAMeTiS, and Jostle-MD are diffusive schemes designed to modify existing partitions to produce a processor allocation. ParMeTiS and Jostle-MS are global partitioners which make no assumptions about the original mesh distribution. Although these partitioners achieve good load balance while minimizing communication overhead, they fail to consider the cost of moving data between processors. A unique feature of PLUM is to address this drawback through the use of an efficient heuristic procedure for redistributing data to assigned processors.

In this study, both communication and remapping costs are optimized by implementing MinEX, that considers computational, communication, and data remapping costs. This feature integrates the communication and remapping cost into the partitioning process. MinEX also redefines the partitioning goal from producing balanced loads to minimizing `MaxQWgt`.

Preliminary experimental results, comparing performance using MinEX to partition the adaptive mesh to those previously obtained from PLUM and SBN, show that MinEX reduces the number of elements migrated by PLUM, and lowers the percentage of edges cut by SBN. For example, for thirty-two partitions with our test case, PLUM showed an edge cut of 10.9% and redistributed 63,270 mesh elements. The corresponding values for the SBN approach were 36.5% and 19,446. Instead, the MinEX partitioner values were 20.9% and 30,548. Note that the MinEX partitioner integrates and optimizes the two important steps of balancing and remapping as part of the partitioning process.

Experiments with MinEX are presented in this chapter to determine the viability of its latency tolerance features on the IPG. The conditions that are required for the IPG to be an effective tool for distributed computations are measured with results showing that MinEX is a viable load balancer provided the nodes of the IPG are connected by a high-bandwidth asynchronous interconnection network.

7.1 Computational test case

Many computational problems are modeled discretely as an unstructured mesh of vertices and edges. To capture evolving features, the mesh topology is frequently adapted. For an efficient parallel implementation, this requires dynamic load balancing. In other words, mesh objects will have to be reassigned after each adaptation phase to rebalance the workload among the processors. It is critical to minimize the overhead associated with remapping data sets, and to reduce the communication between processors at the next solution step. These goals are especially important in an IPG context where communication bandwidths between nodes are likely to be much smaller than on a single multiprocessor machine.

The computational mesh used for the experiments in this chapter simulates an unsteady environment where the adapted region is strongly time-dependent. As shown in section 5.3, a shock wave is propagated through an initial grid to produce the desired effect. The computational mesh is processed through nine adaptations by moving a cylindrical volume across the domain with constant velocity. Grid elements within the cylindrical volume are refined while previously-refined elements are coarsened in its wake. During the processing, the size of the mesh increases from 50,000 elements to 1,833,730 elements.

The data used for the simulations to be presented in this chapter were generated for an Euler solver [74] and uses tetrahedral elements [8] to represent the three dimensional mesh. A dual graph representation of the *original* mesh is used by our experiments for load balancing where each tetrahedra of the original mesh is a vertex of the dual graph. An edge exists between two dual graph vertices if the vertices share a face. Assume that units of computational and communication cost are equal. The description of this dual graph and how it is used in connection with mesh adaptation algorithms was presented in chapter 5. Recall that each vertex of the dual graph has two weights, $\mathtt{PWgt}_v$ and $\mathtt{RWgt}_v$, and one edge weight, $\mathtt{CWgt}_{(v,w)}$, corresponding to edge (v, w). $\mathtt{PWgt}_v$, $\mathtt{RWgt}_v$, and $\mathtt{CWgt}_{(v,w)}$ refer respectively to processing, data remapping, and communication weights associated with processing a dual graph vertex.

Table 16 indicates the scalability of this application where the number of processors, P, is varied from two to 1,024. The data were obtained by simulating the application (details presented in section 7.1.1.) The numbers are non-dimensioned `MaxQWgt` values in thousands. The first row of the table assumes that maximum latency tolerance is achieved; the second row assumes that no latency tolerance is achieved. Maximum latency tolerance is defined as the ability to utilize all available processors to overlap communication and redistribution costs. Table 16 shows that this application can scale linearly to over 128 processors, indicating good potential for an IPG implementation.

Table 16. Scalability analysis of the test application

	Number of Processors									
Latency	2	4	8	16	32	64	128	256	512	1024
Max. Tolerance	3777	1824	1148	614	324	168	89	72	51	57
No Tolerance	4547	3193	1699	1033	558	302	173	123	109	103

Runtimes are presented in thousands of units.

7.1.1 Experimental study

MinEX was executed with actual application data to simulate mesh processing for a variety of system configurations. Individual runs simulates networks with a particular number of processors (P,) number of clusters (C,) `ThroTTle` values, and interconnect slowdowns (I). P was varied from two to 2,048; C was varied from one to eight; `ThroTTle` was varied to find the optimal value for minimizing runtime; and I was varied to simulate high-bandwidth cluster interconnections and low-bandwidth wide area network connections.

Based on performance studies [59, 34], typical communication latency and bandwidth slowdowns from integrated clusters to configurations with clusters connected through a high-bandwidth interconnect are in the range of three to one-hundred. Wide area network connections are one-thousand to ten-thousand times slower than the internal intraconnects of a single cluster. For these experiments, is is assumed that the intra-cluster communication slowdowns to be normalized to a value of unity. Simulations of inter-cluster communication assume slowdown factors of three, ten, one-hundred, and one-thousand. To simplify the analysis, it is assumed that individual processors within a cluster are homogeneous. Processors are also assumed to be divided as evenly as possible among the clusters.

7.1.2 Summary of results

Table 17 and 18 show results of experimental runs analyzing the effect of varying numbers of clusters and interconnect slowdowns. The interconnect slowdowns indicate the slowdown factor relative to intra-communication bandwidth. P is assumed to be thirty-two. To be consistent with results presented in other tables of this chapter, runtimes are shown in thousands of units. All processors are assumed to be homogeneous.

Table 17. Expected `MaxQWgt` with no latency tolerance

	Interconnect Slowdowns			
Clusters	3	10	100	1000
1	473	473	473	473
2	728	863	1228	4102
3	755	1168	2783	18512
4	791	1361	3667	25040
5	854	1649	5677	53912
6	915	1717	8521	76169
7	956	1915	10958	80568
8	968	2178	11492	93566

Runtimes are presented in thousands of units.

The most optimistic expectation of application processing is that the processing activity can entirely hide the data set and communication latency. At the other extreme, the most pessimistic view is that no latency tolerance is achieved. Experiments simulating both views are conducted to analyze the effect of latency tolerance. Table 17 charts the experimental results when no latency tolerance is achieved; table 18 assumes maximum latency tolerance. The following conclusions can be drawn from the results:

- With greater interconnect slowdowns, the runtimes increase dramatically as additional clusters are used. For example, the runtime metric in table 17 is $4,102$ when two clusters and an interconnect slowdown of one-thousand is assumed. The runtime metric is $93,566$ when eight clusters are assumed. The ratio, $93,566/4,102 \approx 22.80$. Considering the interconnect slowdown of three, the ratio between two clusters and eight clusters is $968/728 \approx 1.32$ which is a much smaller value. The same pattern holds true in table 18

- The effectiveness of latency tolerant algorithms compared to algorithms without latency tolerance is determined by measuring runtimes of each approach as the

Table 18. Expected `MaxQWgt` with maximum tolerance

	Interconnect Slowdowns			
Clusters	3	10	100	1000
1	287	287	287	287
2	298	469	763	3941
3	322	548	2386	12705
4	328	680	3297	21888
5	336	768	4369	33092
6	345	856	5044	52668
7	352	893	5480	61079
8	357	1048	5721	61321

Runtimes are presented in thousands of units.

number of clusters and interconnect slowdowns are varied. The relative improvements, from algorithms without latency tolerance to algorithms with latency tolerance, are greater when more clusters are employed. This can be verified by comparing the same rows from table 17 and table 18. For example, consider the row with six clusters. The difference in runtime comparing latency tolerant algorithms to those with no latency tolerance are $915 - 345 = 570, 1,717 - 856 = 861, 8,521 - 5,044 = 3,477, 76,169 - 52,668 = 23,501$ respectively for interconnect slowdowns three, ten, one-hundred, and one-thousand. In contrast, the row with two clusters indicate improvement in latency tolerant runtimes of $728 - 298 = 430, 863 - 469 = 394, 1,228 - 763 = 465, 4,102 - 3,941 = 161$ for the same interconnect slowdowns. In general, the rows that correspond to more clusters show greater runtime improvement when employing latency tolerance. The same cannot be said when analyzing columns of the tables where interconnect slowdowns are varied. For example, the table columns with interconnect slowdowns of one-hundred indicate improvements in $473 - 287 = 186, 1,228 - 763 = 465, 2,783 - 2,386 = 397, 3,667 - 3,297 = 370, 5,677 - 4,369 = 1,308, 8,521 - 5,044 = 3,477, 10,958 - 5,480 =$

$5,478, 11,492 - 5,721 = 5,771$ respectively for clusters one to eight. The columns with interconnect slowdowns of ten show corresponding runtime improvements of $473 - 287 = 186, 863 - 469 = 394, 1,168 - 548 = 620, 1,361 - 680 = 681, 1,649 - 768 = 881, 1,717 - 856 = 861, 1,915 - 893 = 1022, 2,178 - 1,048 = 1,130$. In this case, a clear pattern cannot be established.

- For this mesh application, Globus over low-bandwidth networks such as the Internet is not a viable approach assuming current technology. Low-bandwidth interconnections would have to improve by at least an order of magnitude before the approach could be seriously considered. At present, applications would have to have little runtime communication and data-set remapping for low-bandwidth wide area networks to be practical. However, Globus could be a viable approach if a high-bandwidth interconnect (slowdown factor between three and ten) between clusters is utilized. The first column of tables 17 and 18 comparing one and eight clusters with an interconnect slowdown factor of three respectively show a slowdown factor of 2.04 and 1.24. Similarly, the second column of the tables with an interconnect slowdown factor of ten respectively show slowdown factors of 4.60 and 3.65. These factors being smaller than the number of clusters indicate a speedup from when one cluster of one-eighth the number of processors is used.

- To evaluate the effectiveness of MinEX versus the case where no partitioning was done, additional experiments were executed. Consider the case where the interconnect slowdown factor is one-hundred and four clusters are used. If latency tolerance is employed, the partitioning with MinEX led to reduced runtime improvements from $17,752$ to $3,297$. Similarly, if no latency tolerance is employed, the improvement in runtime units was from $18,323$ to $3,667$. Both show partitioning related improvements in runtime by approximately a factor of five. Other interconnect/cluster combinations show significant improvements as well.

7.1.3 Evaluation of MinEX

This section presented results of experiments that tested the viability of the MinEX partitioner in distributed computing environments. The experiments also measured the conditions that are required for the IPG to be an effective tool for such distributed computations. Results demonstrate that MinEX is a viable load balancer and that the IPG could be effectively used with similar applications provided that the nodes of the IPG are connected by a high-bandwidth asynchronous interconnection network.

7.2 MinEX comparisons to MeTiS

Many application classes, such as those that require solutions to computational unstructured adaptive meshes, require load balancing to be performed while the application executes to prevent processors from becoming either overloaded or idle. Typically, load balancing for this kind of application is accomplished through use of a partitioning program where a graph is supplied. This graph models the processing and communication costs of the application. Many excellent partitioners were developed over the years and a survey of these partitioning approaches is included in [3]. However, the most successful state-of-the-art partitioners are multi-level [41, 46, 77]. Multi-level partitioners contract the supplied graph by collapsing edges, partition the coarsened graph, and then refine the graph back to its original size. Among the fastest and most effective multilevel schemes is MeTiS [46] for which both parallelized and serial versions are available. Variants of MeTis can either partition from scratch or can work diffusively with an existing partition. MeTis produces partitions with low communication cost by minimizing edge cut and can also optimize data movement required to redistribute the workload by minimizing vertex weights that represent the cost of data movement. However, when applied to an IPG environment, MeTiS has some deficiencies that need to be addressed. Examples of these are:

- In a distributed environment, it is possible to achieve a perfectly balanced processing workload and still incur excessive unbalanced redistribution and communication overhead.
- For heterogeneous configurations, processing and communication weights vary. A graph with fixed processing, data mapping, and communication edge weights is not sufficient.
- MeTiS can minimize the edge cut or it can optimize data movement. Unfortunately, it cannot satisfy both constraints in the same invocation. Typically, partitioning and data remapping are accomplished in two separate steps.
- MeTiS cannot anticipate latency tolerance techniques that would be employed by the application to hide the detrimental effect of low bandwidth.

The novel partitioning scheme, MinEX, presented in chapter 6 addresses the limitations of existing partitioners. The advantages of MinEX are itemized in the bullets that follow:

- MinEX has the partitioning goal to minimize the total runtime of the application. This objective is different from that of other partitioners and counters the possibility of achieving balanced loads with excessive communication.
- MinEX utilizes a configuration data structure that models IPG speed parameters such as the number of nodes, the number of processing clusters, and the interconnect slowdowns. The graph being partitioned is mapped onto this data structure to accommodate a highly heterogeneous environment.

- MinEX calls a user supplied function as partitioning decisions are made. This function is used to model the latency tolerance of the application solver and assist in partitioning decisions.

To evaluate MinEX, a solver for the classical N-body problem is implemented and integrated with both the MinEX and MeTiS partitioning schemes. In this application, dynamic load balancing in a multi-computer environment is necessary to optimize efficiency. The N-body solver uses a modified Barnes & Hut algorithm [5] for producing a solution. Configurations of thirty-two processors distributed over up to eight clusters are simulated with interconnect slowdowns varied between three and ten-thousand. Results show that in a simulated IPG environment, MinEX dramatically reduces the runtime requirements of the application as compared to results obtained when using MeTiS.

At this point, the time required to partition a graph is not optimized to the level of MeTiS and the codes are not yet parallelized. Our measurements, therefore, focus on application efficiency and not partitioning time. Specifically, the metrics `LoadImb` and `MaxQWgt`, described in section 5.3.1 will be measured and compared. These metrics are excellent predictors as to how both approaches will perform in a distributed environment. Recall that the motivation for MinEX development is to optimize the partitioning process in such an environment.

This remaining sections of this chapter are organized as follows. Subsection 7.2.1 introduces the N-body problem that is implemented. Subsection 7.2.2 discusses the latency tolerance functions that interface with MinEX. Subsection 7.2.3 describes the experimental study, analyzes the results, and draws conclusions.

7.2.1 The N-body application

The N-body application is the classical problem of simulating the movement of a set of particles based upon gravitational or electrostatic forces. Many applications in the fields of astrophysics, molecular dynamics, and fluid dynamics can utilize N-body solvers. The basic solution involves calculating the velocity and position of each particle, b, at discrete time steps. At each step, there are b^2 pairwise interactions of forces between bodies. Of the many N-body solutions that have been proposed, the Barnes & Hut algorithm [5] is probably the most popular. The approach uses a divide-and-conquer strategy to approximate the force exerted on a body by a cluster of bodies that are sufficiently distant with the center of mass position and total quantity of mass in the cluster. In this way, the total number of force calculations are greatly reduced. The algorithm's first step is to recursively build a tree of cells in which the bodies are grouped by their physical position. By traversing the tree, one can be determine the distance between any pair of cells. Normally the decision as to whether pairs of cells are close or far is made based on dividing the greatest distance between two bodies in a cell by the distance between the center of mass positions of the cells. If the result of the division is greater than a user-defined parameter δ, the cells are considered close and all of the pairwise force calculations are to be performed. An example of a parallel Barnes & Hut implementation using message passing is described in [78] and then later refined in [56]. In this chapter, the Barnes & Hut approach is modified to integrate the MinEX and MeTiS partitioners with the solution.

The application framework

The pseudo code in figure 23 gives an overview of the framework for implementing the N-body application. At each time step adaptation, a new tree is constructed to allocate the N bodies to cells. Before the solver is called, a check is performed to determine whether the processor load is balanced. A partitioner is used to improve load balance if necessary. After the load is balanced, the N-body solver computes the new positions of each of the bodies. The last step of each adaptation is to output statistical and visualization data. These steps are continually processed until all of the adaptations are completed.

```
Output Initial N-body Visualization Data
Do
    Create Barnes & Hut N-body Tree
    If Processor Load is Unbalanced
        Call Partitioner
    Call N-body Solver
    Output N-body Statistics
    Output N-body Visualization Data
Until all adaptations are processed
```

Figure 23. N-body framework.

N-body tree creation

The first step of processing the N-body problem is to recursively build a tree of cells. Starting with an empty tree, the first particle is inserted into an initial leaf cell. Subsequent particles are also placed into this cell until the cell is eventually filled. At this point, an internal cell is allocated with eight pointers for each possible octant. The bodies are then inserted into subtrees at the next level. Figure 24 illustrates this concept showing an internal cell divided into eight geographic octants. Both the spatial and tree representations are shown. The cell's center of mass position is used for subsequent searches of the tree. Traversal direction is determined by the octant where a particle resides relative to this center of mass position In this implementation, up to eight particles can reside in a single leaf cell.

N-body solver

The pairwise force calculations between the particles are calculated using the Newtonian gravitational formulas. For completeness these formulas are itemized:

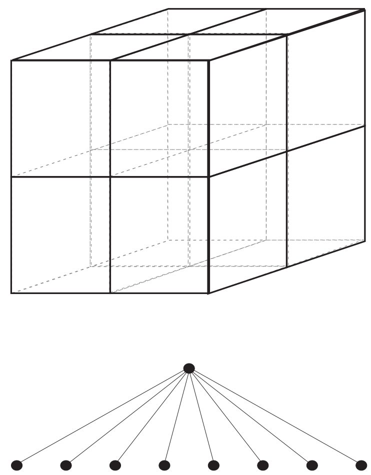


Figure 24. A two level oct-tree.

$< \mathbf{x_b}, \mathbf{y_b}, \mathbf{z_b} >$ is a vector representing the coordinate position of particle, b.

$\mathbf{r_{(b1,b2)}}$ is a scalar representing the distance between particles $b1$ and $b2$ computed as:

$$r_{(b1,b2)} = \sqrt{(x_{b1} - x_{b2})^2 + (y_{b1} - y_{b2})^2 + (z_{b1} - z_{b2})^2)}$$

$\mathbf{F^x_{(b1,b2)}} = \frac{\mathbf{Gm_{b1}m_{b2}(x_{b1} - x_{b2})}}{\mathbf{(r_{(b1,b2)})^3}}$ indicates the gravitational force between bodies in the x direction. G is the gravitational constant and m_{b1} and m_{b2} respectively indicate the masses of particles $b1$ and $b2$. Note that a small smoothing constant is added to $r_{(b1,b2)}$ to prevent division by zero. The gravitational forces between bodies in the y and z directions are similarly defined.

$\mathbf{a_b} =< \mathbf{a^b_x}, \mathbf{a^b_y}, \mathbf{a^b_z} >$ is an acceleration vector computed using the formula $F = ma$. Acceleration is integrated to compute a velocity vector $v_b =< v^b_x, v^b_y, v^b_z >$. A second integration is performed on the velocity vector to compute the particle position $r_{(b1,b2)}$ at the next time step. Integration is performed using a leap-frog method. If p_n, v_n, and a_n respectively indicate position, velocity, and acceleration at time step n, and t is the size of the time step, the leap-frog calculations are:

$$v_{(n+1)/2} = v_n + a_n * t/2, \quad p_{n+1} = p_n + v_{(n+1)/2} * t, \quad v_{n+1} = v_{(n+1)/2} + a_{n+1} * t/2.$$

Processing

The N-body solver is implemented using a message passing model. The pseudo code in figure 25 indicates solver execution at each processor. The processing steps are designed so that the application can minimize the deleterious effects of low bandwidth. The logic proposes a strategy where processors distribute data sets and communication data as early as possible so that processing can be overlapped with communication.

The solver achieves additional latency tolerance by further reducing communication when possible. For example, the position and mass of a particular particle only needs to be communicated once to a remote processor to which a group of close bodies are assigned. Similarly, if far cells are assigned to the remote processor, the center of mass and position of the local cell needs be transmitted only once. Therefore, for each queued particle cell, at most two communications are necessary to each remote processor. A second reduction of communication recognizes that position data need not be received for particles that were relocated from that processor during the same time step. A MinEX calling parameter automatically accommodates this second optimization.

Partitioning

When the tree creation phase of the algorithm is completed, a graph is constructed that is presented to the partitioners. The partitioners utilize this graph to divide the workload among the processors of the grid. Each vertex, v, of the graph corresponds to a leaf cell C_v in the N-body tree and has two weights, $\texttt{PWgt}_v$ and $\texttt{RWgt}_v$. Each edge (v, w) has two weights, $\texttt{CWgt}_{(v,w)}$ and $\texttt{CWgt}_{(w,v)}$. These weights refer respectively to processing, data remapping, and communication costs that will be incurred when the solver processes C_v. The total time required to process the vertices assigned to a processor p must take into account all of these metrics as they are defined in the bullets that follow:

Send all particles that are to be reassigned
For each data set that is relocated to this processor
  Unpack and store the data
  Calculate force interactions with local particles
For each particle assigned to this processor
  Transmit position & mass data
For each particle assigned to this processor
  Calculate force interactions with local particles
For each particle assigned to this processor
  Receive particle communication data
  Calculate force interactions using data received
For each particle assigned to this processor
  Integrate to determine new particle positions

Figure 25. Solver processor pseudo code.

- $\texttt{PWgt}_v$ is the number of computations that are executed by the solver to compute the new positions of the particles residing in C_v. To compute this weight, the following metrics are defined:

  $|C_v|$ represents the number of particles in C_v.

  $\texttt{Close}_v^b$ is the count of particles in cells close to C_v.

  $\texttt{Far}_v$ is the count of cells that are far from C_v.

  Using these metrics,

  $$\texttt{PWgt}_v = |C_v| \times (|C_v| - 1 + \texttt{Close}_v^b + \texttt{Far}_v + 2).$$

  The processing weight $\texttt{PWgt}_v$ includes the pairwise force calculation between the internal $|C_v|$ bodies in C_v, the external particles that are in cells close to C_v, the single force calculations using the center of mass locations of distant cells, and two calculations for the double integration needed to determine velocities and positions of the bodies at the next time step.

- $\texttt{RWgt}_v$ defines the cost of relocating cell C_v from one processor to another. Its weight is set to $|C_v|$ since each of the particles in the cell must be communicated.

- $\texttt{CWgt}_{(v,w)}$ represents the communication cost between cells C_v and C_w. $\texttt{CWgt}_{(v,w)}$ is set to a value of unity if C_w is far from C_v. If C_w is close to C_v, $\texttt{CWgt}_{(v,w)} = |C_v|$. This metric models the amount of communication required between cells.

7.2.2 N-body interface to MinEX

The MinEX partitioner expects the calling program to supply a function to assist in the partitioning and to achieve latency tolerance. This function enables the partitioning to be tailored to the performance of the solver. The user function determines the criteria for a promising vertex move, compares whether one potential vertex move is better to another, controls the order that potential vertex moves are processed, establishes the order in which vertices are contracted, and projects anticipated runtime should a vertex move be executed. The function utilizes a switch statement to enumerate the appropriate sections of code. Each of these enumerated blocks of code are described in this section.

Criteria for promising moves

The N-body user-defined function is called with $\texttt{QLen}$(p), $\texttt{Gain}$, $\texttt{MaxQWgt}$, $\texttt{MinQWgt}$, Δ $\texttt{MinVar}$. These metrics are used to establish the criteria for whether a vertex move should be rejected. The following represent conditions that will reject potential moves:

- A vertex move that will result in a processor having an empty queue ($\texttt{QLen}(p) = 0$.)

- A vertex move with excessive increase in $\texttt{Gain}$. Excessive increase is when $\frac{(\texttt{Gain})^2}{\Delta\texttt{MinVar}} > \texttt{ThroTTle}$ where $\texttt{ThroTTle}$ is a user-defined parameter. In our experience, if $\texttt{ThroTTle}$ is set too low, the partitioner might not achieve a balanced load. On the other hand, if $\texttt{ThroTTle}$ is set too high, the partitioner runs slow and may result in a balanced load but with sub-optimal runtime. In experiments conducted, it was observed that `ThroTTle` generally converges between the values of P and $2P$.

- A move that increases `MaxQWgt`.
- A move with that will not change `MaxQWgt` but will decrease `MinQWgt`.
- If no change is anticipated to either `MinQWgt` or `MaxQWgt`, moves will be rejected that do not significantly reduce Δ `MinVar`. In experiments conducted, Δ `MinVar` was required to be reduced by at least 1 percent.

Evaluating vertex reassignments

MinEX calls the user-defined function to compare pairs of vertex moves. The metrics `QLen`(p), `Gain`, `MaxQWgt`, `MinQWgt`, Δ `MinVar` relating to each move are passed by MinEX for a criteria for comparison to be established. A potential vertex move is considered superior to another if:

- The resulting `MaxQWgt` is lower.
- If `Gain` is negative and the impact upon `MaxQWgt` is equal, the best move will result in the greatest increase in `MinQWgt`.
- When neither `MaxQWgt` or `MinQWgt` are impacted, the vertex move with the lowest resulting `Gain` value is the best.
- If none of the criteria is satisfied, the best move is the one that reduces `MinVar` the most.

Vertex reassignment order

MinEX utilizes a min-heap of potential vertex moves to be processed. The key of this heap controls the order of vertex reassignment. The user-defined function is called with the same metrics as described in the previous two paragraphs. These metrics are used to build the heap-key used in the experiments to be presented in section 7.2.3. The key that is computed is determined by the formula, $(\texttt{GainBias} + \texttt{Gain}) + \texttt{MaxQWgt} \times \texttt{RunBias}$. `GainBias` and `RunBias` are constants that allow for proper scaling of the key. The idea is to prioritize vertex moves first by `MaxQWgt` and then by `Gain`.

Order of graph contraction

Similar to the ordering of processing vertex swaps, the Graph Contraction order is also controlled using a min-heap. Metrics are passed to the user-defined function to provide information relating to edges (v, w) that are to be collapsed. These metrics are $\texttt{PWgt}_v$, $\texttt{RWgt}_v$, $\texttt{CWgt}_{(v,w)}$, $\texttt{CWgt}_{(w,v)}$, and $\texttt{Color}_{(v,w)}$. $\texttt{Color}_{(v,w)}$ is the total number of vertices represented in vertices v and w. Additionally the user-defined function has access to the total number of vertices in the graph, $|v|$, before contraction and the current number of vertices in the graph, $|v_c|$.

The heap-key that is used to control the order of graph contraction is $\texttt{Color}_{(v,w)} \times (|v| - |v_c|) - \texttt{CWgt}_{(v,w)} - \texttt{CWgt}_{(w,v)}$. This order favors collapsing of edges with high edge cuts but tries to minimize the chance that the vertices the original graph are not represented by a small number of meta-vertices.

Latency tolerance

It is the responsibility of the application to determine the percentage of latency tolerance that can be achieved since this is dependent upon the nature of the application solver. This function is called to determine the expected $\texttt{QWgt}(p)$ value that will result at a given processor after a vertex reassignment. Data supplied by MinEX to the latency tolerance function include the metrics $\texttt{Cunpack}_p$, $\texttt{Cpack}_p$, $\texttt{Crcv}_p$, $\texttt{Runpack}_p$, $\texttt{Rpack}_p$, $\texttt{Rrcv}_p$, $\texttt{Pproc}_p$, `QLen`(p), `QSend`(p), and `QOwn`(p) and are are defined in section6.4.8. Another

Metric, $\texttt{QRcv}$(p) that is computed indicates the number of N-body cells to be relocated to processor p. $\texttt{QRcv}(p) = \texttt{QLen}(p) + \texttt{QSend}(p) - \texttt{QOwn}(p)$.

Two sets of calculations are derived to mathematically model the operation of the solver. The first calculations are used to estimate the reduction in communication messages that are achieved by eliminating duplicate transmissions. The second calculations estimate the how much processing can likely be overlapped with communication. These calculations are described in the following paragraphs.

To accommodate the solver's attempt to minimize communication messages, the $\texttt{Cunpack}_p$, $\texttt{Cpack}_p$, and $\texttt{Crcv}_p$ values must be recomputed. For example, the original values of these metrics assume that every N-body cell is communicated to every other N-body cell. It doesn't take into account that the communication need be done only once to another processor to which a queue of cells are assigned. To recompute this value, observe that $\texttt{Cunpack}$ is the total number of unpack operations that are performed in connection with received communications. Before describing the calculations required to recompute $\texttt{Cunpack}_p$, the following terms need to be defined;

$|C|$ is the average number of particles in a cell.

$\texttt{Close}_p$ is the total number of cells received from close cells in remote processors.

$\texttt{Remote}_p$ is the number of cells assigned to remote processors other than p. Note that $\texttt{Remote}_p$ is the difference between total vertices in the partitioning graph and $\texttt{QLen}$(p).

$\texttt{ProbC}_p$ is the probability that a remote cell is close to a cell in processor p. Similarly, $\texttt{ProbF}_p$ is the probability that a remote cell is far from a cell in processor p.

$\texttt{ProbC}_p^x$ is the probability that a remote cell is close to x cells in processor p.

The original $\texttt{Cunpack}_p$ can be determined by the equation:

$$\texttt{Cunpack}_p = \texttt{QLen}(p) \times \texttt{Remote}_p + (|C| - 1) \times \texttt{Close}_p.$$

The first term indicates the total number of separate communication messages received. The second term indicates extra communications needed for each of the particle positions of close cells that are to be transmitted. Using this formula:

$$\texttt{Close}_p = \frac{\texttt{Cunpack}_p - \texttt{QLen}(p) \times \texttt{Remote}_p}{|C| - 1}$$

After solving for $\texttt{Close}_p$: $\texttt{ProbC}_p$, $\texttt{ProbF}_p$, and $\texttt{ProbC}_p^x$ can be determined:

$$\texttt{ProbC}_p = \frac{\texttt{Close}_p}{\texttt{QLen}(p) \times \texttt{Remote}_p}$$

$$\texttt{ProbF}_p = 1 - \texttt{ProbC}_p$$

$$\texttt{ProbC}_p^x = \frac{\texttt{QLen}(p)!}{(\texttt{QLen}(p) - x)!x!}(\texttt{ProbC}_p)^x(\texttt{ProbF}_p)^{\texttt{QLen}(p)-x}$$

The recomputed value of $\texttt{Cunpack}_p$ is:

$$\texttt{Cunpack}_p = \texttt{Remote}_p + \frac{(|C| - 1) \times \texttt{Close}_p\texttt{ProbC}_p^{\texttt{QLen}(p)}}{\texttt{ProbF}_p} + |C| \times \texttt{Close}_\texttt{p} \sum_{i=1}^{\texttt{QLen}(p)} \frac{(\texttt{ProbC}_p)^i}{i \times \texttt{ProbF}_p}$$

Note that the second term subtracts one from $|C|$ because the count of those near cells are already included in the first term. The third term represents those remote cells that transmit both to local cells that are far and also the individual N-body data to those local cells that are near. The division by $\texttt{ProbF}_p$ is needed because all of the $\texttt{Close}_p$ cells are known to be close. Once the revised $\texttt{Cunpack}_p$ value is calculated its ratio to the original value is multiplied by `Crcv` to project its value. Observe that a similar analysis as just described is used to recompute $\texttt{Cpack}_p$.

To estimate the amount of overlap between communication and processing first observe that it is not possible to hide communication with the processing associated with packing and unpacking ($\mathtt{Cpack}_p$, $\mathtt{Cunpack}_p$, $\mathtt{Rpack}_p$, and $\mathtt{Runpack}_p$.) The solver performs force calculations as N-body data are remapped that relate to the incoming particles and the local close bodies. The number of cells involved in these calculations is be estimated by:

$$\{2\mathtt{QRcv}(p) \times (\mathtt{QOwn}(p) - \mathtt{QSend}(p)) + \mathtt{QRcv}(p)(\mathtt{QRcv}(p) - 1)\} \times \mathtt{ProbC}_p$$

The ratio between this and total cell processing, $\mathtt{QLen}(p) \times \mathtt{Remote}_p$, is multiplied by $\mathtt{Pproc}_p$ to predict the amount of processing accomplished during data distribution. The remaining amount of processing can be overlapped with data communication. If $\mathtt{HideR}_p$ and $\mathtt{HideC}_p$ represent the respective amounts of processing hidden by data redistribution and communication, the total estimate of $\mathtt{QWgt}(p)$ is:

$$\mathtt{QWgt}(p) = \mathtt{Cpack}_p + \mathtt{Cunpack}_p + \mathtt{Rpack}_p + \mathtt{Runpack}_p + Max \begin{cases} \mathtt{Rrcv}_p + 1 \\ \mathtt{HideR}_p \end{cases} + Max \begin{cases} \mathtt{Crcv}_p + 1 \\ \mathtt{HideC}_p \end{cases}$$

Using the estimates for latency tolerance just derived, actual solver operation was projected to within 5 to 10 percent in experiments that were conducted.

7.2.3 Experimental study

In the experimental study that is presented in this chapter, a message passing environment is simulated in which the actual N-body solver is executed and where the results of two adaptations are averaged. The test case consists of 1,024 bodies. By varying the grid definition file, the parameters are varied that predict performance on a variety of grid configurations. Individual runs simulate networks with the varying number of clusters ($Clusters$) and interconnect slowdowns (I). To simplify the analysis in our experiments, the number of processors is set to a value of thirty-two and individual processors are assumed to be homogeneous and divided evenly among from one to eight clusters.

Table 19. Expected runtimes (MeTiS)

	Interconnect Slowdowns				
Clusters	3	10	100	1000	10000
1	16	16	16	16	16
2	16	23	91	825	8228
3	16	23	109	1017	10152
4	16	23	119	1115	11141
5	16	23	123	1161	11601
6	16	23	128	1205	12046
7	16	23	131	1244	12431
8	16	23	132	1253	12526

Runtimes are presented in thousands of units.

Based on performance studies [59], typical communication latencies and bandwidth slowdowns from integrated clusters to configurations with clusters connected through a high-bandwidth interconnect are in the range three to one-hundred. Wide area network connections are one-thousand to ten-thousand times slower than the internal intraconnects of a single cluster. To predict performance on a wide range of possible grid environments, we have run experiments with inter-communication slowdown factors of three, ten, one-hundred, one-thousand, ten-thousand. Note that the intra-cluster communication slowdowns are assumed to be normalized to a value of unity.

The results of partitioning with the N-body application and the MeTiS partitioner are shown in tables 19 and 21. Similarly, results for experiments using MinEX are shown in tables 20 and 22. Tables 19 and 20 plot project runtimes in thousands of units. Tables 21 and 22 plot the load balance that was achieved during the same experiments. Recall that load balance (`LoadImb`) is computed using the formula `MaxQWgt/WSysLL`. Additionally, note that load balance relates to values actually achieved by the running of the application. It does not indicate the actual projected imbalance by the partitioners

Table 20. Expected runtimes (MinEX)

	Interconnect Slowdowns				
Clusters	3	10	100	1000	10000
1	15	15	15	15	15
2	15	15	40	304	3645
3	15	15	38	331	3386
4	16	15	51	372	3695
5	15	15	52	396	4115
6	16	16	74	391	4360
7	15	16	46	393	4315
8	16	15	70	405	4090

Runtimes are presented in thousands of units.

before the application was executed. The following conclusions can be drawn from the results in tables 19 through 22:

On fast interconnect systems (slowdown factor of three,) MeTiS and MinEX performance is comparable, although MinEX has a slight advantage in both resulting runtimes and load balance. The same conclusion can be stated for single cluster systems as represented by the top rows in the tables.

Observing the second column of the tables (slowdown factor of ten,) notice that the application using MinEX was able to completely hide the effects of the slower bandwidth. MeTiS, on the other hand showed a 50 percent performance drop when compared to the results shown in the first column of the table 19. This demonstrates that the latency tolerance features of MinEX are effective.

Looking at the results of third to fifth columns (slowdown factors of one-hundred, one-thousand, and ten-thousand respectively,) observe that the advantages of MinEX over MeTiS increase dramatically. For example, with eight clusters and a slowdown factor of ten-thousand, MinEX performance relative to MeTiS shows a $12,526,416/4,390,389 =$

Table 21. Expected load balance (MeTiS)

	Interconnect Slowdowns				
Clusters	3	10	100	1000	10000
1	1.07	1.07	1.07	1.07	1.07
2	1.07	1.07	2.08	2.49	2.37
3	1.07	1.08	2.09	2.23	2.24
4	1.07	1.08	2.07	2.18	2.18
5	1.07	1.08	2.05	2.14	2.14
6	1.07	1.08	2.04	2.12	2.12
7	1.07	1.08	2.09	2.17	2.17
8	1.07	1.08	2.02	2.09	2.10

Table 22. Expected load balance (MinEX)

	Interconnect Slowdowns				
Clusters	3	10	100	1000	10000
1	1.03	1.03	1.03	1.03	1.03
2	1.04	1.04	1.17	1.22	1.26
3	1.04	1.06	1.10	1.16	1.14
4	1.04	1.03	1.28	1.11	1.10
5	1.03	1.03	1.24	1.12	1.10
6	1.05	1.06	1.52	1.07	1.16
7	1.04	1.06	1.16	1.09	1.12
8	1.05	1.03	1.37	1.05	1.12

3.06 times improvement. MinEX also demonstrates superior resulting load balance than what is shown for MeTiS (1.12 percent versus 2.10 percent in tables 22 and 21.)

Both MinEX and MeTiS show a trend towards a slight loss of performance as more clusters are added. The loss is insignificant with smaller slowdown factors (less than 1 percent for slowdown factors three and ten.) With larger slowdown factors the loss still relatively minor (18 percent with a slowdown factor of ten-thousand and MinEX used.) This demonstrates that configurations with more clusters of less processors per cluster can be viable for running applications that are programmed to maximize latency tolerance.

7.2.4 Conclusions

Using a solver that was developed for the N-body experiments conducted, the performance of MinEX was compared to MeTiS, a popular state-of-the-art multi-level partitioner. Results show that while MinEX produces partitions of only slightly better quality to MeTiS on single cluster configurations; on grid configurations with low-bandwidth communication, partitions produced by MinEX yield application runtimes that can be up to three times faster. These results demonstrate the benefits and feasibility of mapping the multiprocessor configuration graph onto the graph to be partitioned and incorporating latency tolerance into the partitioning process; they also illustrate the suitability of MinEX for operation in grid environments.

CHAPTER 8

DISCUSSION AND FUTURE RESEARCH

In this dissertation, three portable topology-independent dynamic load balancing algorithms were presented based on Symmetric Broadcast Networks (SBN). These three SBN-based algorithms were experimentally compared to other poplular dynamic balancers (e.g., Random, Gradient, Sender Initiated, Receiver Initiated, Adaptive Contracting, and Tree Walk) with results showing that the new algorithms compare favorably. The SBN Basic algorithm was further modified to work with adaptive mesh applications where dynamic load balancing algorithms are not often used. By establishing a global view of the dynamic mesh, these adaptations prove that SBN load balancers can be effectively used with this application class and have the advantage of reducing data set remapping, although larger communication costs are incurred. This trade-off is acceptable for applications dominated by a high cost of data set redistribution.

SBN portability is established by its successful execution on two state-of-the art commercial machines (the IBM SP2 and the SGI Origin 2000) without any changes to any of the code being necessary. SBN's topology independence was also illustrated by implementing a variant of the Basic algorithm that is specifically designed for the hypercube. Further utilization of reflective gray code mappings whould allow the SBN approach to be easily ported to other architectures.

Another contribution is the SBN Heuristic Variant algorithm which, based on a mathematical model, achieves a significant reduction in runtime communication over the Standard and Hypercube variants.

A novel partitioner, called MinEX, has also been introduced in this dissertation which is designed specifically for a distributed runtime environment such as NASA's Information Power Grid (IPG). Results comparing MinEX to the fast and state-of-the-art MeTiS partitioner conclude that MinEX achieves superior quality partitions and reduces application runtime requirements in such an environment.

Areas of future research are to optimize and parallelize the MinEX codes so that the runtime required for partitioning is competitive with MeTiS. Subsequently, the MinEX partitioner will be packaged to encourage a widespread distribution and acceptance. A further study of the relationship between the `MinVar` and `Gain` metrics discussed in section 6.4.2 should result in practical conclusions that will simplify the use of the partitioner and improve its efficiency. Investigation of latency tolerance techniques will also be helpful to facilitate the implementation of more applications on geographically separated distributed computing environments. Another area of related research is to develop a general purpose tool that can be used to quickly simulate IPG environments by varying heterogeneity and wide area interconnect parameters.

BIBLIOGRAPHY

[1] D. Abramson, R. Sosic, J. Giddy, and R. Hall, "Nimrod: A tool for performing parameterized simulations using distributed workstations," *4th IEEE Symposium on High Performance Distributed Computing*, (1995).

[2] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias, and M. Snir, "SP-2 system architecture," *IBM Systems Journal*, Vol. 34 (1995).

[3] C. Alpert and A. Kahng, "Recent directions in netlist partitioning," *Integration, the VLSI Journal*, Vol. 19 (1995), pp. 1–81.

[4] S. Barnard, R. Biswas, S. Saini, R. Van der Wijngaart, M. Yarrow, and L. Zechtzer, "Large-scale distributed computational fluid dynamics on the Information Power Grid using Globus," *7th Symposium on the Frontiers of Massively Parallel Computation*, (1999), pp. 60–67.

[5] J.E. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force calculation algorithm," *Nature*, Vol. 324 (1986), pp. 444–449.

[6] R. Biswas, S.K. Das, D.J. Harvey, and L. Oliker, "Parallel dynamic load balancing strategies for adaptive irregular applications," *Applied Mathematical Modeling*, Vol. 25:2 (2000), pp. 109–122

[7] R. Biswas and L. Oliker, "Experiments with repartitioning and load balancing adaptive meshes," *Grid Generation and Adaptive Algorithms*, IMA Volumes in Mathematics and its Applications, Springer-Verlag, Vol. 113 (1999), pp. 89–111.

[8] R. Biswas and R.C. Strawn, "A new procedure for dynamic adaption of three-dimensional unstructured grids," *Applied Numerical Mathematics*, Vol. 13:6 (1994), pp. 437–452.

[9] R. Biswas and R.C. Strawn, "Mesh quality control for multiply-refined tetrahedral grids," *Applied Numerical Mathematics*, Vol. 20:4 (1996), pp. 337–348.

[10] H. Casanova and J. Dongarra, "NetSolve: A network server for solving computational science problems," *Technical Report* CS-95-313, University of Tennessee, 1995.

[11] W. Chan and A. George, "A linear time implementation of the reverse Cuthill-McKee algorithm," *BIT*, Vol. 20 (1980), pp. 8–14.

[12] N. Chrisochoides, "Multithreaded model for the dynamic load balancing of parallel adaptive PDE computations," *Applied Numerical Mathematics*, Vol. 20 (1996), pp. 321–336.

[13] T.H. Cormen, C.E. Lieserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw Hill, 1989.

[14] J. Cryzyk, M. Meznier, and J. More, "The network-enabled optimization system (NEOS) server," *Preprint* MCS-P615-0996, Argonne National Laboratory, 1996.

[15] G. Cybenko, "Dynamic load balancing for distributed-memory multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 7:2 (1989), pp. 279–301.

[16] S.K. Das and S.K. Prasad, "Implementing task ready queues in a multiprocessing environment," *Proceedings of the International Conference on Parallel Computing*, (1990), pp. 132–140.

[17] S.K. Das, S.K. Prasad, C-Q. Yang, and N.M.Leung, "Symmetric broadcast networks for implementing global task queues and load balancing in a multiprocessor environment," *Technical Report*, CRPDC-92-1, Department of Computer Science, University of North Texas, 1992.

[18] S.K. Das, D.J. Harvey, and R. Biswas, "Adaptive load-balancing algorithms using Symmetric Broadcast Networks: performance study on an SP2," *Proceedings of the International Conference on Parallel Processing*, Bloomingdale, IL, (1997) pp. 360–367.

[19] S.K. Das, D.J. Harvey, and R. Biswas, "Parallel processing of adaptive meshes with load balancing," *27th International Conference on Parallel Processing*, Minneapolis, Minnesota, August 10–14, 1998, pp. 502–509

[20] S.K. Das, D.J. Harvey, R. Biswas, and L. Oliker, "Portable parallel programming for the dynamic load balancing of unstructured grid applications," *International Parallel Processing Symposium*, (1999), pp. 338–342.

[21] T. Defanti, M.D. Brown, and R. Stevens, "Virtual reality over high-speed networks," *IEEE Computer Graphics and Applications*, Vol. 16 (1996), pp. 42–43.

[22] T. Defanti, I. Foster, M. Papka, R. Stevens, and T. Kuhlfuss, "Overview of the I-Way wide area visual supercomputing," *International Journal of Supercomputer Applications*, Vol. 10 (1996), pp. 123–130.

[23] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann, "Remote engineering tools for the design of pollution control systems for commercial boilers," *International Journal of Supercomputer Applications*, Vol. 10 (1996), pp. 208–218.

[24] T. Disz, M. Papka, M. Pellegrino, and R. Stevens, "Sharing visualization experiences among remote virtual environments," *International Workshop of High Performance Computing for Computer Graphics and Visualization*, Springer-Verlag, (1995), pp. 217–237.

[25] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, Vol. SE-12:5 (1986), pp. 662–675.

[26] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Evaluation*, Vol. 6 (1986), pp. 53–68.

[27] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser, "Active Messages: A mechanism for integrated communication and computation," *Proceedings of the 19th International Symposium on Computer Architecture* (1992).

[28] M.R. Eskicioglu, "Design issues of process migration facilities in distributed systems," *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, (1995), pp. 414–424.

[29] C. Farhat, "A simple and efficient automatic FEM domain decomposer," *Computers and Structures*, Vol. 28 (1988), pp. 579–602.

[30] J.E. Flaherty, R.M. Loy, C. Ozturan, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. Ziantz, "Parallel structures and dynamic load balancing for adaptive finite element computation," *Applied Numerical Mathematics*, Vol. 26 (1998), pp. 241–263.

[31] I. Foster and C. Kesselman, *The Grid Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.

[32] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, Vol. 11:2 (1997), pp. 115–128.

[33] I. Foster, J. Insley, G. Von Laszewski, C. Kesselman, M. Thiebaux, "Wide-area implementation of the Message Passing Interface," *Parallel Computing*, Vol. 24:12 (1998), pp. 1735–1749.

[34] I. Foster and N. Karonis, "A grid-enabled MPI: Message passing in heterogeneous distributed computing systems," *Proceedings of the Supercomputing Conference*, (1998), on line http://www-fp-mcs-anl.gov/ foster/papers.html.

[35] G. Fox, A. Kalawa, and R. Williams, "The implementation of a dynamic load balancer," *Proceedings on Conference of Hypercube Multiprocessors*, (1987), pp. 114–121.

[36] http://science.nas.nasa.gov/Groups/Tools/IPG/.

[37] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[38] *The Globus Project*, http:://www.globus.org

[39] A. Grimshaw, W. Wulf, and the Legion team, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, Vol. 40 (1997), pp. 39–45.

[40] G. Heber, R. Biswas, P. Thulasiraman, and G.R. Gao, "Using multithreading for the automatic load balancing of adaptive finite element meshes," *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel* (1998).

[41] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," *Technical Report* SABD83-1391M, Sandia National Laboratories, Albuquerque, 1993.

[42] G. Horton, "A multi-level diffusion method for dynamic load balancing," *Parallel Computing*, Vol. 19 (1993), pp. 209–229.

[43] W.E. Johnston, D. Gannon, W.J. Nitzberg, and W. Van Dalsem, "Information power grid implementation plan," *Working Draft*, NASA Ames Research Center, 1999.

[44] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of Parallel and Distributed Computing*, Vol. 48:1 (1998), pp. 71–95.

[45] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, Vol. 20:1 (1998), pp. 359–392.

[46] G. Karypis and V. Kumar, "Parallel multilevel K-way partitioning scheme for irregular graphs," *Technical Report* 96-036, Deptartment of Computer Science, University of Minnesota, Minneapolis, 1996.

[47] B.W.Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, Vol. 49 (1970), pp. 291–307.

[48] S. Khuri and A. Baterekh, "Genetic algorithms and discrete optimization," *Methods of Operations Research*, Vol. 64 (1991), pp. 133–142.

[49] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vechhi, "Optimization by simulated annealing," *Science*, Vol. 220 (1983), pp. 671–680.

[50] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M. E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1993.

[51] G.A. Kohring, "Dynamic load balancing for parallelized particle simulations on MIMD computers," *Parallel Computing*, Vol. 21 (1995), pp. 683–693.

[52] P. Krueger and M. Livny, "The diverse objectives of distributed scheduling policies," *Proceedings of the International Conference on Distributed Computing Systems*, (1987), pp. 242–249.

[53] J. Leigh, A. Johnson, and T. DeFanti, "CAVERN: A distributed architecture for supporting scalable persistence and interoperability in collaborative virtual environments," *Virtual Reality Research, Development and Applications* Vol. 2 (1997), pp. 217–237.

[54] F.C.H. Lin and R.M. Keller, "The gradient model load balancing method," *IEEE Transactions on Software Engineering*, Vol. SE-13 (1987), pp. 32–38.

[55] M. Litzdow, M. Livny, and M.W. Mutka, "Condor — A hunter of idle workstations," *8th International Conference of Distributed Computing Systems*, (1988), pp. 104–111.

[56] P. Liu and S. Bhatt, "Experiences with parallel N-body simulations," *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1988, 122–131.

[57] R. Luling, B. Monien, and F. Ramme, "Load balancing in large networks, a comparative study," *Proceedings of the Symposium on Parallel and Distributed Processing*, Dallas, TX, (1991), pp. 686–689.

[58] Message Passing Interface Forum, "MPI: Message-Passing Interface Standard, Version 2," *Technical Report*, University of Tennessee, Knoxville, 1997.

[59] S. Nog and D. Kotz, "A performance comparison of TCP/IP and MPI on FDDI, fast Ethernet, and Ethernet," *Technical Report* PCS-TR95-273, Dartmouth College, 1996.

[60] L. Oliker and R. Biswas, "PLUM: Parallel load balancing for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, Vol. 52:2 (1998), pp. 150–177.

[61] L. Oliker, R. Biswas, and H.N. Gabow, "Performance analysis and portability of the PLUM load balancing System," *Euro-Par'98 Parallel Processing*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1470 (1998), pp. 307–317.

[62] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving finite element equations on concurrent computers," *Parallel Computations and their Impact on Mechanics* (1986), pp. 209.

[63] S. Pulidas, D. Towsley, and J. A. Stankovic, "Embedding gradient estimators in load balancing algorithms," *Proceedings of the International Conference on Distributed Computing Systems*, (1988), pp. 482–490.

[64] V. Sarkar and J. Hennessy, "Compile-time partitioning and scheduling of parallel programs," *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, (1995), pp. 61–70.

[65] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *Journal of Parallel and Distributed Computing*, Vol. 47 (1997), pp. 109–124.

[66] K.G. Shin and Y.C. Chang, "Load sharing in hypercube multicomputers for real-time applications," *Proceedings of the Conference on Hypercubes, Concurrent Computers, and Applications*, Vol. 1 (1989), pp. 617–621.

[67] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and load balancing in parallel and distributed systems*, IEEE Computer Society Press, 1995.

[68] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, Vol. 25 (1992), pp. 33–44.

[69] W. Shu and M.-Y. Wu, "Runtime incremental parallel scheduling (RIPS) on distributed memory computers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7:6 (1996).

[70] Origin2000 *Technical Report*, Silicon Graphics Inc., Mountain View, 1997.

[71] H.D. Simon, *Computing Systems in Engineering*, Vol. 2 (1991), pp. 135–148.

[72] A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi, "Fine-grain multithreading with the EM-X multiprocessor," *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures* (1997), pp. 189–198.

[73] R.C. Strawn, R. Biswas, and M. Garceau, "Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise," *Journal of Aircraft*, Vol. 32:4 (1995), pp. 754–760.

[74] Strawn, Biswas, and Garceau, "A finite-volume Euler solver for computing rotary-wing aerodynamics on unstructured meshes," *Journal of the American Helicopter*, Vol. 38:2 (1993), pp. 61–67.

[75] R. Van Driessche and D. Roose, "Load balancing computational fluid dynamics calculations on unstructured grids," *Parallel Computing in CFD*, AGARD-R-807 (1995), pp. 2.1–2.26.

[76] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, "Parallel dynamic load balancing algorithm for three-dimensional adaptive unstructured grids," *AIAA Journal*, Vol. 32 (1994), pp. 495–505.

[77] C. Walshaw, M. Cross, and M.G. Everett, "Parallel dynamic graph partitioning for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, Vol. 47 (1997), pp. 102–108.

[78] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree N-body algorithm," *Supercomputing '93*, 1993, pages 12-21.

BIOGRAPHICAL INFORMATION

The author received his doctorate in computer science engineering from the University of Texas at Arlington in August 2001.

EDUCATION

Ph.D. in Computer Science & Engineering, UT Arlington, 2001
Thesis: Load Balancing Techniques for Distributed Processing Environments.

M.S. in Computer Science, Fairleigh Dickinson University, 1974
Thesis: Analysis of IBM Software Packages.

B.S. in Mathematics, Queens College, 1968.

PROFESSIONAL EXPERIENCE

Assistant Professor Dallas Baptist University, 1991–2001.
Visiting Research Consultant NASA Ames Research Center, 1999.
Independent Software Consultant Micro Plus Inc., 1990–91.
Adjunct Professor Sussex County Community College, 1990–91.
President XDS, Inc. and Digital Methods Inc., 1978–90.
Senior Analyst Sweda International, 1975–1978.
Senior Systems Programmer Dunn and Bradstreet, 1973–75.
Systems Programmer Bell Labs, 1972–73.
Systems Programmer Singer Kearfott, 1969–72.

RESEARCH INTERESTS

Load Balancing, Parallel Algorithms, Distributed Computing, Parallel Architectures.

AWARDS AND ACTIVITIES

Finalist for Best Paper Award, *IEEE 2001 International Symposium on Cluster Computing and the Grid*, (with R. Biswas and S. K. Das).

Nominated for Outstanding Doctoral Research Student at UTA, 2001.

Head Cross Country and Track Coach for six years at Dallas Baptist University.
Completed five marathons including the 101st running of the Boston Marathon in 1997.

Member of the ACM, USATF, NAIA, NCAA, and USTCA; Level 2 USATF certification.

PUBLICATIONS BY AUTHOR

1. A latency-tolerant partitioner for distributed computing on the Information Power Grid, *CD-ROM proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, California, (2001), pp. 23–27. (with R. Biswas and S. K. Das).
2. Latency hiding in dynamic partitioning and load balancing of grid computing applications, *International Symposium on Cluster Computing and the Grid (CCGrid)*, Brisbane Australia, (2001), pp. 347–354 (with R. Biswas and S. K. Das).
3. Parallel processing of adaptive meshes with load balancing, *IEEE Transactions on Parallel and Distributed Systems*, (2001), to appear (with R. Biswas and S. K. Das).
4. Adaptive load-balancing algorithms using Symmetric Broadcast Networks, *Journal of Parallel and Distributed Computing*, under review (with R. Biswas and S. K. Das).
5. Parallel dynamic load balancing strategies for adaptive irregular applications, *Applied Mathematical Modeling*, Vol. 25:2 (2000), pp. 109–122 (with R. Biswas, S. K. Das, L. Oliker).
6. Portable parallel programming for the dynamic load balancing of unstructured grid applications, *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Computing*, Puerto Rico, (1999), pp. 338–342 (with R. Biswas, S. K. Das, and L. Oliker).
7. Dynamic load balancing for adaptive meshes using Symmetric Broadcast Networks, *12th International Conference on Supercomputing*, Melbourne, Australia, (1998), pp. 417–424 (with R. Biswas and S. K. Das)
8. Design of novel load-balancing algorithms with implementations on an IBM SP2, *3rd International Euro-Par Conference*, Passau, Germany, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1300 (1997), pp. 937–944 (with R. Biswas and S. K. Das)
9. Parallel processing of adaptive meshes with load balancing, *27th International Conference on Parallel Processing*, Minneapolis, Minnesota, (1998), pp. 502–509 (with R. Biswas and S. K. Das).
10. Adaptive Load-balancing algorithms using Symmetric Broadcast Networks: Performance Study on an IBM SP2 *26th International Conference on Parallel Processing*, Bloomingdale, Illinois, (1997), pp. 360–367 (with R. Biswas and S. K. Das).
11. Adaptive load-balancing algorithms using Symmetric Broadcast Networks, *NASA Ames Research Center Technical Report*, NAS 97-014, (1997) (with R. Biswas and S. K. Das).
12. Performance analysis of an adaptive Symmetric Broadcast load balancing algorithm on the hypercube, *Department of Computer Science*, University of North Texas, Technical Report CRPDC-95-1 (1995) (with S. K. Das).