UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Programming Model for Automated Decomposition on Heterogeneous Clusters of Multiprocessors

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science in
Computer Science

by

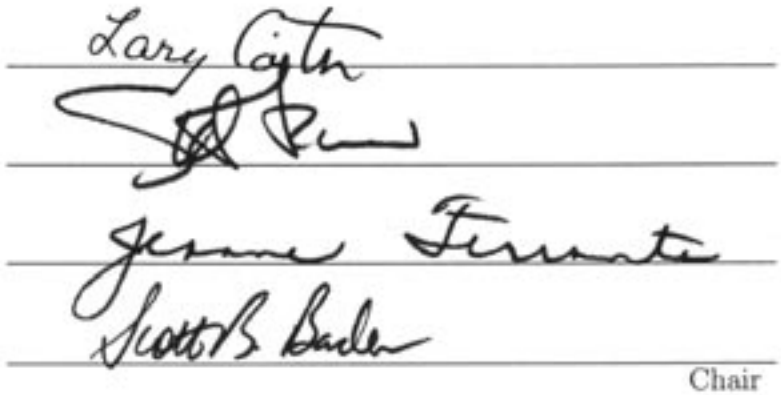
Sean Philip Peisert

Committee in charge:

Professor Scott B. Baden, Chair
Professor Larry Carter
Professor Jeanne Ferrante
Professor Sidney Karin

2000

The thesis of Sean Philip Peisert is approved, and it is acceptable in quality and form for publication on microfilm:



Chair

University of California, San Diego

2000

Dedicated to:

My parents and grandparents, who raised me wonderfully and gave me everything I needed to figure out how to succeed in life, including my Opa, who is no longer here to see me present this thesis, but who first set me in front of a typewriter – an experience that without, I would probably never have taken to the computer, much less the supercomputer.

"It is an old maxim of mine which states that once you have eliminated the impossible, whatever remains, however improbable, must be the truth."

– Sherlock Holmes, *The Sign of Four*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank the following people and institutions for their generous support and encouragement during my life and the research and writing process of this thesis:

All my friends, especially my best friend Stephen Shore for many enlightening discussions about life, the universe, everything, and beyond, for the past twelve years that helped shape who I am and bring me to this point. And for a lot of fun. And to Alex, Eric, Kent, Laura, P.J., Steve, and Tage, too. I have such fantastic friends!

My girlfriend Cathy who has given me inspiration when things proved to be most frustrating elsewhere.

Ms. Jean Isaac, with whom I had my first computer class in the First Grade at St. Mark's School in Marin County, California, who inspired me to explore computers more.

Mrs. Judy Farrin, my great friend and teacher, who made computers and chemistry at Redwood High School in Marin County, California, fun and interesting by believing in me and letting me explore on my own. And for letting me blow things up in the chemistry lab.

Dr. Paul H. J. Kelly, Dr. Jarmo Rantakokko, and Dr. Daniel Shalit for their help and friendship, especially when things were going very awry.

Uppsala University in Sweden for generously allowing me use of the Yggdrasil DEC Alpha cluster.

The National Center for Supercomputing Applications (NCSA) and their extremely knowledgeable and helpful staff for the extremely generous allocation of computing time on the Silicon Graphics Origin2000 machines there and their help in using them. When I couldn't find any other machines to use, NCSA heroically bailed me out. In particular, I would like to thank Faisal Saied, Louis Hoyenga, Susan John, Yiwu Chen, Roy Heimbach, Scott Koranda, Dave McWilliams, and Michael Pflugmacher.

UCSD and the UCSD Computer Science and Engineering department for five-and-a-half great years of college and graduate education and a very good time.

The San Diego Supercomputer Center (SDSC) and the National Partnership for Advanced Computing Infrastructure (NPACI) and so many people there. I certainly can't list every person who has been so wonderful in their friendship and support, but some of the people who have been there most have been Jeanne Aloia, Harry Ammons, Mike Bailey, Chaitan Baru, Gracie Cheney-Parsons, Cheryl Converse-Rath, Sandy Davey, Richard Frost, Jon Genetti, Anke Kamrath, Sid Karin, Greg Johnson, Dan Jonsson, Amit Majumdar, Yuki Marsden, Jennifer Matthews, Steve Mock, Reagan Moore, Arcot Rajasekar, Wayne Schroeder, Teri Simas, Allan Snavely, Ken Steube, Shawn Strande, Peggy Wagner, and Bill Zamora. Without my first internship at SDSC doing MacSupport, I never would have become involved with supercomputers and be where I am today.

Everyone in the PCL, especially Shelly Strout for her friendship and making writing a thesis more fun (and helping to keep me sane).

My fantastic professors, especially Professor Larry Carter, with his enlightening, challenging, interactive classes that made algorithms and performance programming fun.

Finally, Professor Scott Baden, my friend and thesis advisor, who gave me the opportunity to explore parallel and scientific computation and write a Master's thesis. Despite many other life experiences, I had never done anything quite like this. It's been a unique, fun, life-changing experience that has given me not just an interest scientific computing but an interest and enthusiasm towards academia and research in general.

This work was supported by UC MICRO program award number 99-007 and Sun Microsystems.

VITA

March 23, 1976	Born, Marin County, California
1993–1996	Intern, Autodesk, Inc.
1996–1997	Freelance Writer, TidBITS
1997	Study Abroad, Oxford University, England
1997	Programming Intern, Pixar Animation Studios
1998	Programming Intern, Dolby Laboratories, Inc.
1999	B.A. University of California, San Diego Minor in Organic Chemistry
1999	Research Assistant, University of California, San Diego
1996–2000	Programming Intern, San Diego Supercomputer Center (SDSC)
2000	M.S., University of California, San Diego
1994–Present	Computer Consultant
2000–Present	Chief Operating Officer, Dyna-Q Corporation

PUBLICATIONS

"Simulating Neurotransmitter Activity on Parallel Processors." S. Peisert, S. Mock, and S. B. Baden, UCSD Research Review 1999 Poster Session, February 26, 1999.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Computer Graphics.
Dr. Michael J. Bailey

Studies in High-Performance and Scientific Computing.
Professor Scott B. Baden

Studies in George Gershwin.
Dr. Elizabeth B. Strauchen,
Wolfson College, Oxford University, England

Studies in Russian Literature.
Dr. Roger D. Dalrymple,
St. Peter's College, Oxford University, England

ABSTRACT OF THE THESIS

A Programming Model for Automated Decomposition on
Heterogeneous Clusters of Multiprocessors

by

Sean Philip Peisert
Master of Science in Computer Science
University of California, San Diego, 2000
Professor Scott B. Baden, Chair

Clusters of multiprocessor nodes are becoming common in scientific computing. As a result of the expandability of clusters, faster nodes are frequently added and older nodes are gradually removed, making the cluster heterogeneous. As heterogeneity increases, traditional methods for programming clusters of multiprocessors become less optimal, because they do not account for the fact that a cluster will only run as fast as the slowest node. Sputnik is a programming methodology and software library that addresses the problem of heterogeneity on a dedicated cluster of multiprocessors.

Sputnik uses a two-stage process for running applications on a cluster of multiprocessors. The first stage assesses the relative performance of each node by running the program individually on each node, determining from the run times both the performance and application-specific optimization. Using the timings obtained from stage one, the second stage partitions the dataset non-uniformly, according to the relative speed of each node. All future runs of the program use the optimal partitionings and number of threads per node.

Sputnik is implemented on top of the KeLP infrastructure to handle irregular decomposition and data motion. It enables code to be written for a heterogeneous cluster as if the cluster is homogeneous. Sputnik can run scientific

applications on a heterogeneous cluster faster, with improved utilization, than a nearly identical program written in KeLP alone. Experimental results from a pair of SGI Origin2000's indicate that Sputnik can improve run-time of an iterative solver for Poisson's equation by 35 percent.

Chapter I

Introduction

I.A Motivation

The computer hardware used for scientific and technical computing is continually evolving. In the most recent generations of supercomputing hardware, there have been *vector supercomputers*, made by companies including Cray as well as *multicomputer*-style *massively parallel processors (MPPs)* made by many companies, including Cray, IBM, Intel, Hewlett-Packard and SGI. *Clusters of multiprocessors*, however, are increasing in popularity, replacing older mainframes. As a result of mass-production of the components used, monetary costs for purchasing clusters of multiprocessors are dropping and therefore, use of the technology has been spreading from business computing to scientific and technical computing, replacing vector supercomputers and multicomputer MPPs in science where they replaced mainframes in industry.

Unfortunately, although multiprocessors and multiprocessor clusters are attractive to organizations with a need for large-scale parallel computation, well-established techniques used to program multicomputer MPPs and vector machines are not always the optimal techniques to program multiprocessors or multiprocessor clusters. Further, since one of the appealing aspects of clusters of multiprocessors is that many of their components can be built from readily-available, commer-

cial hardware solutions, such as Sun, IBM or SGI multiprocessor workstations, it can be cost-effective to add in or swap out systems in the cluster at will, replacing old components gradually. The result is that what was originally a *homogeneous* cluster of multiprocessors can easily become *heterogeneous* over time, with the addition of newer systems with different processor speeds, number of processors, memory sizes, cache sizes and network speeds, an example of which is shown in figure I.1.

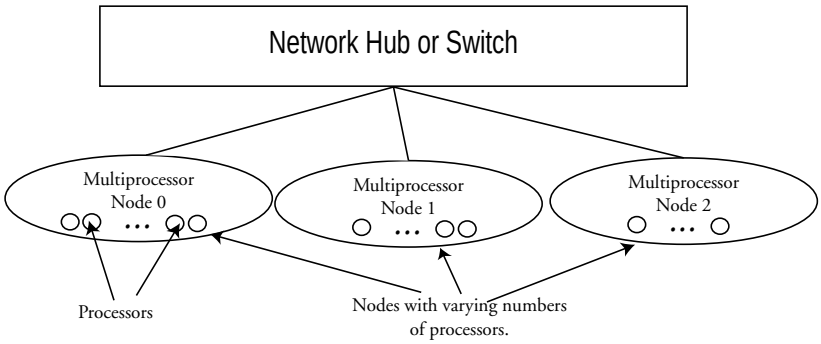


Figure I.1: Diagram of a heterogeneous cluster of multiprocessor nodes

In a heterogeneous cluster, a uniform partitioning of data across the cluster is not optimal because some of the nodes will finish before others, leaving parts of the cluster idle until the slower nodes terminate. This problem can generally be stated to say that *a cluster will only run as fast as the slowest component node.* Thus, to the degree that any node finishes early and idles while waiting, we will under-utilize the hardware and nodes will be forced to idle rather than spending their time productively computing. Ideally, therefore, we want all nodes to finish at the same time.

Though the techniques for programming multiprocessors and homogeneous clusters of multiprocessors have been explored in detail and have achieved some level of sophistication, programming heterogeneous clusters of multiprocessors for use with scientific computation is still a difficult challenge [8][9][10][11][12][13][14][23][24][25][26][27][35][42]. Existing software technologies may be

used to program heterogeneous clusters of multiprocessors, however, the process of doing so and still achieving good performance through *load-balancing* can be extremely difficult.

The goal of my research presented in this thesis is to investigate a way to enable scientific programs to run faster and effectively utilize a heterogeneous cluster of multiprocessors while allowing the user to write the program as if they were running on a homogeneous cluster.

This thesis introduces an API called *Sputnik* designed to assist in greater performance and utilization on a heterogeneous cluster for certain types of applications. Current applications for which Sputnik has been proven to function well with include stencil-based programs. Applications with extremely finely-grained communication might be inappropriate for the current iteration of the API because extremely tight communication imposes a kind of synchronization on the program that might negate the speedup that Sputnik can provide.

The API is packaged as a C++ library and is built on top of the KeLP infrastructure [23][35]. The library allows users to write scientific programs that run effectively on *heterogeneous* clusters of multiprocessors where the component nodes may all run at different speeds. It is intended as something of a "proof of concept" about what steps are needed to make heterogeneous clusters run more efficiently. I also present a broader theory of which the API is merely a subset. The broader theory, the *Sputnik Model*, is not limited to multiprocessors, stencil-type applications, or an just two optimization techniques.

This thesis introduces a two-stage process for optimizing performance on a heterogeneous cluster. Though Sputnik has been targeted for multiprocessors, some or all of the nodes may be uniprocessors.

The first stage, the *ClusterDiscovery* stage, performs a *resource discovery* to understand how the application, or perhaps a part of the application, runs on each individual node in the cluster. The second stage, *ClusterOptimization*, makes specific optimizations based on what the first stage has discovered. Depending on

the hardware and the type of problem, there can be many possible types of optimizations. In this thesis, I make one specific application study with two different types of optimizations: repartitioning the amount of data each node works on and adjusting the number of threads that run on each node.

This thesis does not attempt to solve the problem of scheduling heterogeneous clusters of multiprocessors that are networked over a heavily-trafficked wide-area network, including *grids*. Such problems are best solved through different methods, including dynamic load-balancing by use of the Globus, Legion, Network Weather Service or AppLeS [30][39][66][69]. Blending one or more of these technologies with my API is beyond the scope of this thesis.

The thesis also does not attempt to tackle the problem of machines with a deeper hierarchy than two tiers (processor and node). A deeper hierarchy might be a "cluster of clusters." Additionally, this thesis and the API it presents are specifically focusing on clusters of multiprocessors. It is not looking at the added level of detail of what happens when several completely different architecture types are clustered together, such as vector and multicomputer-style MPP supercomputers, and parts of the computation run better on different architectures.

Optimizations including the ones I have just mentioned, in addition to many others, are possible optimizations that could be done in ClusterOptimization. A process whereby the ClusterOptimizer does nothing more than vary the tiling of the problem to fit in level 1 or level 2 cache of varied sizes is certainly also possible. The emphasis is, however, that the ClusterDiscovery and ClusterOptimization are separate processes that can function in tandem easily and are not just limited to multiprocessors. Other architectures and possible optimizations are beyond the scope of this thesis and are not addressed in this incarnation of the API that I have developed.

Finally, I make certain assumptions about the condition of the cluster:

1. Multiprocessor nodes are connected by a uniform, local, dedicated network.

2. The program running has dedicated access to the cluster hardware.

3. The cluster is set up to have many more processors per node, on average, than nodes in the whole cluster. A cluster with many nodes of very few processors (including ASCI Blue Pacific) may be better off with a single-tiered approach including MPI because the shared memory aspect of Sputnik will be much less relevant.

4. No node in the cluster runs less than half as fast as any other node in the cluster.

5. The problem does not fit, in entirety, into memory cache on any of the processors.

I.B Heterogeneity

I take the concept of some of the existing API's for programming clusters of multiprocessors one step further by directly supporting heterogeneous clusters. Acting on the assumption that clusters of multiprocessors are the immediate future of high-performance parallel computing, I decided that programming clusters of multiprocessors was a problem worth investigating.

I decided that I could build my API on top of either two existing API's — one that could handle message-passing between nodes and one that could handle shared-memory communication within a node — or one existing API that already supported multi-tier communication.

In choosing a basis for my API, I decided to use KeLP1 as my message-passing layer because, unlike MPI and PVM, it has built-in support for data description and general blocked decomposition as well as transparent message-passing communication. This makes the job of the programmer using the API much easier for complex parallel programming [23][35]. Such levels of abstraction have been shown to come without a performance penalty [11]. The question then was what

technology would be best used to take advantage of the shared-memory architecture for intra-node communication. I use OpenMP as my shared-memory layer to handle intra-node parallelization because at the moment, it is the easiest technology to use and is an emerging standard as an alternative to Pthreads. My API is built on top of KeLP and OpenMP. KeLP itself is built on top of MPI. OpenMP, can be based on a variety of sub-technologies, depending on the particular vendor's implementation. The Origin2000, which I run on, uses SGI's "sprocs," though the IBM SP systems build OpenMP on top of Pthreads [48].

Hopefully by extending KeLP1's API instead of simply MPI's, I have not only introduced a method for improved performance, but reduced the complexity to do multi-tier programming.

Most importantly, Sputnik targets heterogeneous clusters. As clusters of multiprocessors age, unless a cluster is completely replaced, at high cost, as opposed to being upgraded, being able to utilize an entire cluster without having an entire node or parts of several nodes remain idle is desirable. No programmer, researcher, or owner of the cluster will want to waste precious time on a costly, high-maintenance piece of computing hardware. To that end, Sputnik does irregular partitionings and regulates the amount of OpenMP threads that are used within each node.

I.C Organization of the Thesis

This thesis discusses the structure of and problems associated with programming a heterogeneous cluster of multiprocessors.

Chapter 2 presents background information on multiprocessors, clusters of multiprocessors and tools for programming both.

Chapter 3 introduces a variety of methods for programming clusters of multiprocessors and also discusses a variety of experiments that I ran which led up to my conclusion that the API I had in mind would work.

Chapter 4 discusses the most important aspect of this thesis and my research — the theory of operation of the Sputnik Model. In addition, chpater 4 also looks at a few implementation details.

Chapter 5 discusses the results of the Sputnik API in an application study.

Chapter 6 presents my conclusions along with future work possibilities.

The appendices contain source code and present a user's guide to the Sputnik API.

Chapter II

Clusters of Multiprocessors

II.A Multiprocessors

A *multiprocessor* is a machine with two or more processors that all share the same main memory, as shown in Figure II.1. Some multiprocessors contain processors that are equidistant from the main memory. This is referred to specifically as a *symmetric multiprocessor* or *SMP* which has *uniform-memory access* or *UMA*. By contrast, some multiprocessors, including the SGI Origin2000, have internal networks which cause the processors to have *non-uniform-memory access* or *NUMA* because the amount of time it takes for a processor to retrieve memory from two other processors might differ. The SGI Origin2000 actually has a NUMA variation called *cache-coherent NUMA* or *ccNUMA*. Typically although main memory is shared on a multiprocessor, each processor in the node is separated from main memory by one, two or sometimes even three levels of individual cache.

Multiprocessors are starkly different from their vector supercomputer and massively parallel multicomputer ancestors. First, by definition, a multiprocessor uses *shared memory*, whereas in an *multicomputer*, all processors have their own, separate main memories, making them *distributed-memory* (also called *shared nothing*). (The SGI Origin2000, the principle machine used in obtaining the re-

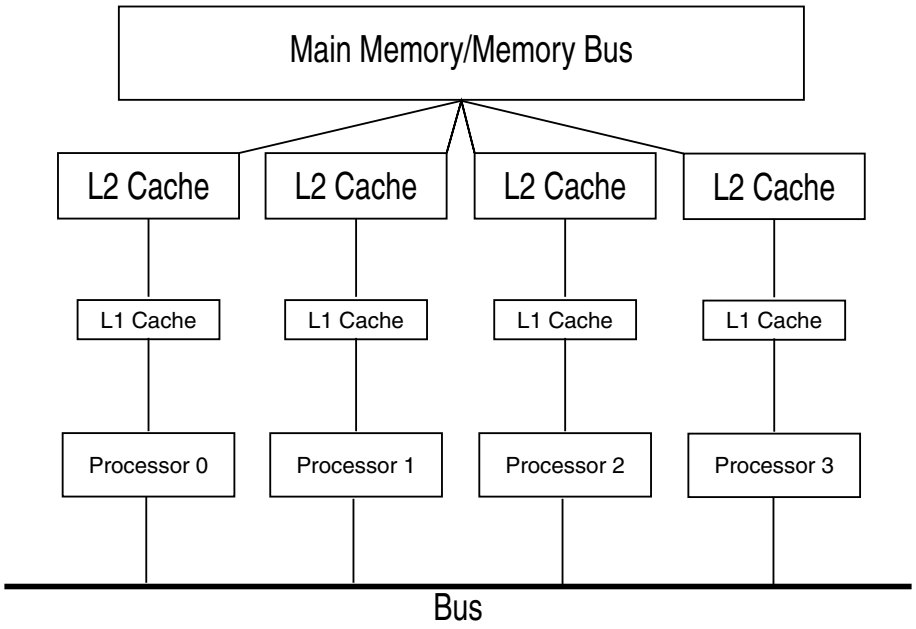


Figure II.1: Diagram of a multiprocessor

sults presented in this thesis might be considered a hybrid between the multiprocessor and multicomputer because it uses *distributed shared-memory.*) Second, the multiprocessors that compete in today's market with modern multicomputers are typically built partially from commodity parts for the purpose of reducing cost by means of increased economies of scale. A recent multiprocessor incarnation from IBM, for instance, forms the basis of each node comprising the the ASCI Blue Pacific machine at Lawrence Livermore National Labs and Blue Horizon at the San Diego Supercomputer Center [37][62]. These machines use hundreds of IBM's PowerPC processors - a chip family that powers all modern Apple Macintosh computers [33][37][38]. Similarly, the chip inside Sun's Enterprise servers, the Sparc, also powers every Sun workstation that comes off the line [67][68]. The large IBM systems, despite being massively parallel, are still essentially large clusters of multiprocessors. As discussed below, many multicomputers are built using expensive, specialty communications hardware significantly more complex than that of an multiprocessor.

In a multiprocessor, parallel programs can typically be run either using message passing across the bus that connects the processors or by interacting using their shared memory. The latter is often accomplished by way of using a threads library such as Pthreads (short for POSIX Threads — a POSIX-compliant thread library) that supports mutual exclusion on pieces of memory so that one thread can hold exclusive access to a piece of memory while it is writing to that piece so that other threads get only the final value when the thread that has the lock is finished writing to it [6][15][50][51][52]. The API calls for Pthreads are starkly different from MPI. Instead of "communication primitives," there are commands that do thread manipulation (e.g. `create`, `fork` and `join`) and commands that do memory "locking" so that no more than one thread has access to a critical section of code simultaneously (e.g. `pthread_mutex_lock` and `pthread_mutex_unlock`). A program running on a multiprocessor using shared memory can often achieve significantly better timing results because it is specifically using the shared memory hardware that a multiprocessor is designed to use, rather than a congested processor bus.

It is possible to simulate message-passing with shared-memory and also possible to simulate shared-memory with message-passing. Machines that are called *distributed shared memory*-type machines are often examples of the latter. A distributed shared memory machine is one that would have a library that would allow "shared memory"-style calls even though the memory may or may not necessarily be physically shared on the hardware. This means that one processor might be able to access the memory of a processor on a completely different machine because the software has been built to allow that style of access. This is called a *non-uniform memory access* or *NUMA*.

One of the principle challenges in future development of multiprocessors is determining what an optimal configuration is. Due to the nature of the construction of a multiprocessor, it is possible that putting too many processors in one multiprocessor could cause congestion on the bus. Additionally, too many

threads all trying to perform read and write accesses to the same place in memory can create a bottleneck due to each process having to wait for other process's mutex locks to unlock before they can go and set their own mutex lock. [21][44][45]

II.B Multicomputers

In parallel programs where one processor works on data and then needs to exchange data with another processor, there must be communication between processors. In a *shared nothing* architecture (distributed-memory machine without shared memory), the only way to do this is to pass *messages* between the processors. In message passing, one processor communicates with other processors through a basic set of message-passing primitives, including *send*, *receive*, *broadcast*, *scatter*, and *gather*. Using send and receive, one processor sends a message across the communications network while the other receives it. In broadcast and scatter methods, one processor sends a message to all other processors in the network. In the gather method, all processors send to a single processor in the network. It is clear that as more messages are being passed, the more congested the network becomes and the more complex the solutions needed to solve the problem of building a low-*latency*, high-*bandwidth* network.

An example of a distributed memory machine can be seen in Figure II.2. An extremely basic example of a simple distributed-memory machine might use a *bus* to pass messages. The problem with this particular design is that the bus that messages travel on is often a bottleneck and would therefore not scale well to larger number of processors due to competing demands on the bus. For this reason, large machines typically use a more scalable interconnect.

The advantages of topologies like the *crossbar*, the hypercube, or the toroidal mesh, as is used in the SGI-Cray T3E, is that it makes for an extremely fast network and is very expandable to a large numbers of processors. Unfortunately, toroidal meshes or crossbars are among technologies that are very expensive to

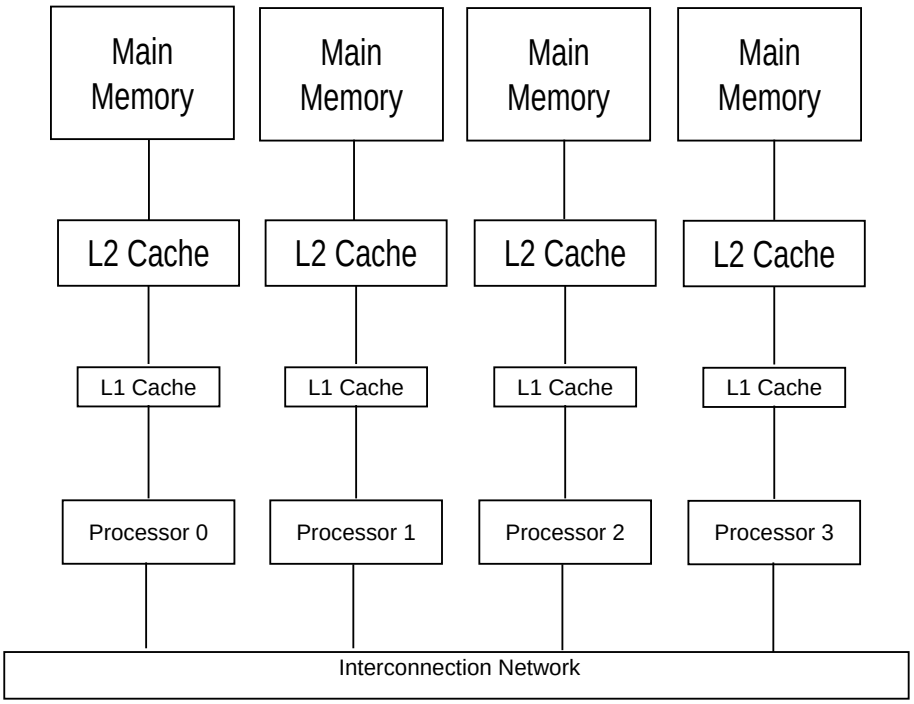


Figure II.2: Diagram of a distributed-memory machine

construct. Using a crossbar switch, for instance, would involve connecting every processor to every other processor directly. As the number of processors grows, the number of connections grows by:

$$\frac{p(p-1)}{2} \tag{II.1}$$

Number of Processors	Number of Connections
2	1
3	3
4	6
5	10
6	15
7	21

Table II.1: Growth of connections in a crossbar-switched machine

II.C Clusters of Multiprocessors

A cluster of multiprocessors is the composite of two distinct hardware technologies for building parallel machines: multicomputers and multiprocessors.

A cluster of multiprocessors is a possible solution to the lack of expandability of a lone multiprocessor and the expensive nature of expanding a multicomputer. This solution has been adopted in high-end server lines and even supercomputers from nearly all major computer hardware manufacturers including Sun, IBM, HP, and Compaq/DEC. SGI has created a machine called the "Origin2000" which uses shared memory but has a very advanced hypercube interconnection structure to support scalability of up to 256 processors per Origin2000. An image of what the Origin2000's architecture looks like can be seen in figure II.3. Some of the computers with the highest theoretical peak speed in the world, such as ASCI Red at Sandia National Labs (Intel), ASCI Blue Pacific at Lawrence Livermore National Labs (IBM) and the Blue Horizon machine at the San Diego Supercomputer Center (IBM) are all essentially clusters of multiprocessors. [33][62]

Clusters of multiprocessors are also sometimes called *hierarchical* machines or *multi-tier* machines because one level of communication is possible within the node, generally shared-memory, and one level of communication is possible between nodes themselves. The two tiers of processor and node-level communication is why they are considered to be hierarchical. It is certainly possible to take the model beyond two tiers as well, to three or more, by connecting a two-tiered cluster of multiprocessors to another two-tiered cluster of multiprocessors, thus creating a three-tiered cluster of multiprocessors, or a cluster of clusters of multiprocessors. Clusters of more than two tiers are outside of the scope of this thesis.

Clusters of multiprocessors are attractive because the manufacturers are able to scale the systems to large number of processors much easier and more inexpensively than if they tried to solve the problem of networking every processor together (as in a multicomputer) or share memory between all processors (as in an

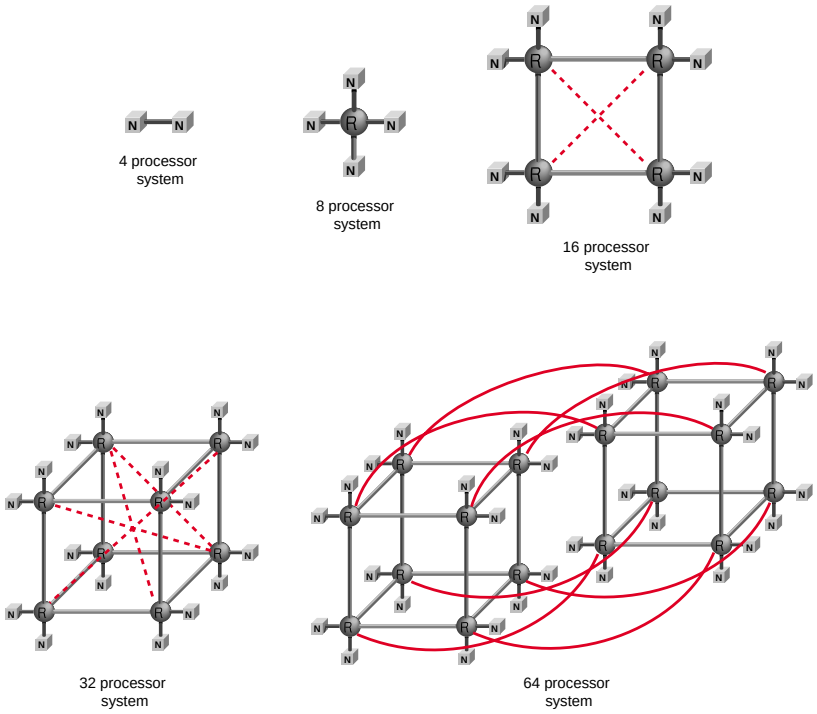


Figure II.3: Diagram of an SGI Origin2000 (Courtesy of SGI's "Origin2000 and Onyx2 Performance Tuning and Optimization Guide (IRIX 6.5)")

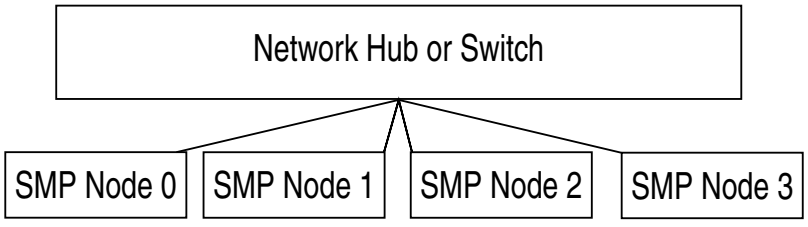


Figure II.4: Diagram of a cluster of symmetric multiprocessors

multiprocessor).

The principal downside of a cluster of multiprocessors is that they are difficult to program. Memory within a node is only "shared" between processors within a given multiprocessor node. Therefore, a shared memory program cannot be used across the entire cluster. Message passing can be used across the entire cluster, but this does not take advantage of the unique shared-memory hardware that exists within a multiprocessor, which is is what often makes communication in a multiprocessor fast. So message passing throughout the entire cluster is not optimal either.

The solution currently being adopted to program clusters of multiprocessors is to use a dual-tier approach of combining message passing and shared-memory programming, as if blending multicomputer and multiprocessor programming techniques, respectively. In this manner, within a multiprocessor (*intranode*), communication can be achieved through shared-memory (Pthreads, OpenMP) and between multiprocessor nodes (*internode*), communication can be achieved through message-passing (e.g. MPI and PVM)[15][50][52][51][6]. One can either hand-write the code that combines one method from each paradigm or use a library that specifically supports multi-tier machines, including KeLP2 or SIMPLE [35][23][14]. Since programming a piece of software with MPI in mind can easily double the size of the code, and programming and debugging in Pthreads can be a challenging task as well, the prospect of combining both technologies to form a highly-tuned piece of parallel code can be daunting. For this reason, using a library with multi-tier support built in can make programming an application significantly easier than when using its component technologies, MPI and Pthreads — by hand — without sacrificing performance [23]. I will discuss various programming libraries and methods in the next chapter.

Chapter III

Heterogeneous Multi-Tier Programming

III.A Previous Work

Significant progress has already been made in programming homogeneous multi-tier machines, but the issue is still an open problem. Software APIs have been developed which can specifically address a multi-tier machine. In the simplest approach, at least one vendor has implemented MPI on their machines so that if a message is to go to another processor on-node, it gets converted into a native shared memory call. If it goes off-node, it gets converted into a native messaging-layer call. Finally, if it goes off-machine, it is sent via TCP/IP. A standardized technology exists based on this concept called Multiprotocol Active Messages and additionally Sun has implemented similar technology in their version of MPI that runs on the Sun Enterprise 6500 and Enterprise 10000 servers [42]. Other vendors are working on their own implementations.

III.A.1 Multi-Protocol MPI

Multi-protocol active messages are a technology that have been developed to allow different messaging protocols depending on where the message is in the machine. If the message needs to be passed within an SMP, for instance, the message is "passed" using shared memory. Within an MPP, the message is passed using a native messaging layer. Outside the MPP, between MPPs, the message is passed using TCP/IP. Lumetta finds that the shared memory protocol can achieve five times the bandwidth of the networking protocol, for instance.

While multi-method or multiprotocol communication, especially when running across MPI, can theoretically run effectively on a heterogeneous cluster, there is also a strong possibility that it may underperform an API that has been specifically designed in a multi-tier fashion. The reason for this is that although the multiprotocol-capable API can use all of the processors and even take advantage of whatever special shared memory and networking hardware exists in the cluster, it is entirely possible that the runtime system will distribute the MPI processes in an nonoptimal manner that requires significantly more communication than is necessary. An API that has been specifically designed to partition on two levels, as the machine itself is built on a hardware level, can specify exactly how all the processes should be arranged. For example, the MPI processes may be scattered throughout the entire cluster when the program executes. Some processes might be able to share memory within the cluster, but there is no guarantee that those processes even need to communicate with each other. One wants to make sure, for instance, that the processes or threads are as close by as possible to the others that they communicate with. Whereas a multi-tier API can ensure an order to the distribution of the processes or threads, a scattering of MPI processes cannot. Culler [42], Fink & Baden [24][26], and Fink [23] employ higher-dimensional partitionings to solve this problem.

Finally, and *most importantly*, although multiprotocol MPI can use all the processors and thus make some heterogeneous clusters function as if they are

homogeneous, this is only true when all processors are the same speed. If the processors are all different speeds, then merely utilizing all the processors and the shared memory hardware is not enough. One needs to make sure that slower processors are processing less data.

Although multiprotocol MPI is reasonable for a programmer to use to quickly run existing MPI-based parallel software on a cluster of multiprocessors and realize improved performance, starting from the ground up with a multi-tier program written in Sputnik (or migrated from KeLP to Sputnik), the programmer can avoid the problems that I have just mentioned.

III.A.2 Multi-Tier APIs

SIMPLE

SIMPLE is an API that is based on two lower-level technologies [14] and primarily addresses collective communication. One is a message passing layer and the other is an SMP node layer. The principle requirement is that the message passing library implements a collection of eleven message passing primitives and the node library implements three shared-memory primitives. These combine to implement a final set of eight SIMPLE primitives (`barrier`, `reduce`, `broadcast`, `allreduce`, `alltoall`, `alltoallv`, `gather`, and `scatter`. Although MPI (including MPICH) works as a messaging layer, the architects of SIMPLE discovered that the Internode Communication Library (ICL) provided superior performance. They also decided to use Pthreads as the SMP node layer, although as they describe, a faster library, possibly something vendor-specific, might work even better.

The calls in the SIMPLE library allow the programmer to work across multiple SMPs seamlessly instead of having to partition the dataset twice manually – a primary partition to determine which data segment runs on each SMP and a secondary partition to determine which part of each data segment runs on each individual processor of each SMP.

KeLP2

Expanding upon the KeLP1 API that already made programming SMPs easier with support for region calculus and data motion, among other parallel programming abstractions, KeLP2 supports the unique multi-tier structure of clusters of SMPs [23][26][12][8]. While KeLP1 aids programmers in making parallel code easier to write without suffering performance penalties, KeLP2 does the same thing for multi-tier machines. Essentially, KeLP2 opened up a whole new and more powerful class of machines for parallel programmers to use the KeLP-style technology on.

Whereas KeLP1 was an interface on top of the MPI [6] technology underneath, KeLP2 was an interface on top of both MPI and Pthreads, as shown in Figure III.1. As described previously, the idea was that although MPI could be used for both inter- and intra-node communication, in theory, this is not the optimal communication method because MPI does not utilize the shared-memory hardware. Instead the designers of KeLP2 decided that MPI would be used for inter-node communication and Pthreads would communicate within an SMP node using fast, fine-grained, shared-memory accesses that Pthreads are particularly good for. KeLP2 adds support the multi-tier nature of a cluster of SMPs, if the SMP nodes are different, however, one sees inefficiencies. The program will only run as fast as the slowest node and so if one node finishes in 13 seconds and another in 5, the program will take 13 seconds to run.

Although KeLP2 is a very good multi-tier implementation, at the time of writing the Sputnik API, it was not running on my platform of choice, the Origin2000, so I built my Sputnik API on top of KeLP1 (MPI) instead of KeLP2 (multi-tier with MPI and Pthreads).

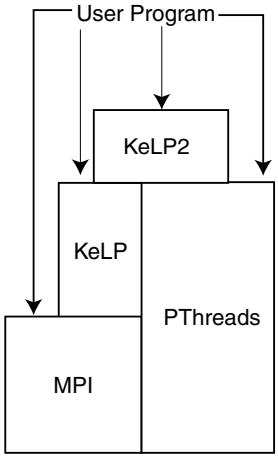


Figure III.1: Hierarchy of software layers for KeLP2

III.A.3 Heterogeneity Work

Grid-Enabled MPI

In Grid-Enabled MPI, Foster attempts to reduce communication time inherent in sending large, frequent communications over a network with high contention and varying latencies. He proposes that collective MPI operations, such as MPI_Reduce, "might well first reduce within each SMP node, then within each MPP, and finally across MPPs." [28]

Non-Uniform 2-D Grid Partitioning

Crandall investigates different partitioning schemes for heterogeneous computing [19][20]. Unlike the Sputnik API, which does a one-dimensional decomposition, Crandall's work suggests a multi-dimensional decomposition using a variety of different schemes including block, strip and "Fair Binary Recursive Decomposition." The advantage of this work, over a plain one-dimensional strip decomposition, is that the cache-miss rate can theoretically be improved. In a simple block decomposition, for instance, by adjusting the dimensions of the block's edges, one can attempt to fit the rows (or columns, depending on the programming language

used) in cache, thus improving the performance by reducing the number of expensive cache misses occurring. Also, the amount of data that needs to be transmitted, but not necessarily the amount of sends and receives, can be constrained. Crandall claims that this trade-off can lead to an overall savings in communication time. Whereas strip decomposition might be extremely straightforward, irregular blocks can be much more complicated, however.

Crandall also works with a "decomposition advisory system" which functions like Sputnik in some respects, choosing an optimal decomposition system based on pre-known aspects of the computational demands.

Zoom

Anglano, Schopf, Wolski and Berman investigated a method for describing heterogeneous applications in terms of structure, implementation and data. The motivation is that not every machine existing (e.g. multicomputers, vector supercomputers, multiprocessors) is adept at solving all problems. The Zoom representation attempts to solve this problem by allowing the user to abstract the program such that each particular segment in the abstraction can be sent to the machine or class of machine that it is best suited for [5]. Such a technology would be a fantastic improvement to Sputnik in the future. See the Proceedings of the Heterogeneous Computing Workshop (1995) and Proceedings of the 1992 Heterogeneous Workshop (IEEE CS Press) for more information on related technology.

III.B Heterogeneous Multi-Tier Programming

III.B.1 Problem

1. As discussed previously, *the goal of my research, presented in this thesis, is to find a way to make scientific programs run faster and improve utilization on heterogeneous clusters of multiprocessors and still allow the user to write*

the program as if they were running on a homogeneous cluster.

2. The *problem* is that they currently have imbalanced loads if running homogeneous partitions on heterogeneous multiprocessor nodes.

3. Therefore, the *solution* is to partition the dataset in a way that the nodes would each finish their runs at the same time.

4. The *problem* of how to partition the dataset to do this was then created.

5. The *solution* to the partitioning, I decided, was to find the relative speeds of each multiprocessor node and calculate the fraction of the power of the whole cluster that each individual multiprocessor node had. Then, one would assign the same fraction of the dataset to a node as the fraction of power of the node has in the whole cluster. "Power" for a node is defined to be the inverse of the time that a node takes to run a benchmark relative to the sum of the inverses of the timings of the same benchmark on every node in the cluster.

III.B.2 Requirements

As I have discussed, existing programs that use a hybrid of a message-passing and shared-memory model as tools to program multi-tier machines [1], do not work effectively on heterogeneous machines. The feasibility studies I made and how they were modified, in some cases, from existing programs, to support heterogeneous clusters, follow in the next section.

An MPI-based messaging library was an obvious choice to use as the inter-node messaging component. The reason is that MPI is standardized across all of the multiprocessor platforms, with each vendor creating their own implementation adhering to the MPI standard [6]. Not only does each vendor specifically have a

[1]The currently recommended method for many current cutting-edge clusters of multiprocessors, including the IBM Blue Horizon machine at the San Diego Supercomputer Center and ASCI Blue Pacific at Lawrence Livermore National Labs [37][62].

finely-tuned implementation of MPI for running optimally on their hardware, but there are freely available versions of MPI as well, including MPICH [7].

Of further interest beyond MPI, however, was to be able to use a message-passing library that supports both heterogeneity and block decomposition and also assists in hiding most of the details of heterogeneity, especially data decomposition. KeLP is a library that supports these requirements and has its communication mechanisms built on top of MPI [10].

The experiments that lead up to my combination of OpenMP — a thread library easier to use than Pthreads that is programmed with compiler directives and API calls — and KeLP follow.

The first question I attempted to answer in the following experiments was whether or not the idea of repartitioning the data helped address the heterogeneous cluster. I did this by first working with a benchmark built with MPI and Pthreads combined to work on a multi-tier machine. The second question I wanted to answer was whether MPI and OpenMP — the two underlying technologies that I ultimately wanted to use — were interoperable. At the same time, I wanted to assess the scalability of OpenMP on the Origin2000. Finally, I tested KeLP and OpenMP together to make sure that by using KeLP instead of MPI (even though KeLP is built on top of MPI) there were no new incompatibilities of performance bugs when used with OpenMP.

III.C Multi-Tier Experiments

III.C.1 MPI and Pthreads

In order to determine whether a heterogeneous API for KeLP might work, I decided to use a hand-coded multi-tier program in a simulated, heterogeneous hardware environment. To do this, I used a piece of software called "Red-Black 3D" (hereafter referred to as rb3D or redblack3D) [23][11]. According to its own documentation: "The rb3D program is a 3D iterative solver which solves Poisson's

equation using Gauss-Seidel's method with Red/Black ordering [58]." The program is described in more detail in chapter 5.

This particular implementation of the rb3D algorithm is hand coded using MPI and Pthreads. This implementation partitions the data twice — once for the node level and once for the processor level. MPI is used to pass messages between multiprocessors and Pthreads are used to communicate within a multiprocessor node via shared memory. The Pthread model is different than the one OpenMP uses in many ways, but one important issue is what happens to the threads during the run of the program between iterations. Pthreads uses "parked threads," meaning that instead of the threads being destroyed between iterations or sections of the program, they are temporarily parked for future use. The benefit of this is that there is less time overhead in continually creating and destroying threads. The downside is that the threads are using system resources when other system processes might need a spare moment on a CPU.

Instead of a simple, static, uniform partitioning where each node n of N is assigned the fraction n/N of the work, my version uses two partitioning schemes. The first is an optimized, non-uniform partitioning, based the partitioning on the total number of processors, as opposed to nodes. It calculates the total number of processors available on all nodes, P, the total number of nodes, N, and how many processors each individual node has, $p_0 \cdots p_{n-1}$. It then assigns work chunks of the total data set, W, equal to the fraction of processing power each node has, $w_0 \cdots w_{n-1} = (p_0 \cdots p_{n-1})/P$. Therefore, as long as everything is equal in a machine, especially the processing power of each individual processor, except the number of processors in a node, the problem workload will remain balanced and the problem will run optimally fast based on the configuration of the cluster of multiprocessors.

One cannot be guaranteed an optimal problem as in the first partitioning scheme, however. More than likely, speeds of processors will differ, speeds of memory and networks will differ, and cache sizes and other important machine

characteristics will differ. Therefore, by having the user issue the proper flag to the program to initiate balanced, non-uniform partitioning, and have a previously-generated file containing the timing results prepared, the program can partition based on any arbitrary data. The idea is that the program would be run on each individual multiprocessor, the times would be recorded, in order, and put into the file. Then, in the future, the program could be run without re-running the resource discovery part of the program.

The results of this implementation are shown in figure III.2 and table III.1. As can be seen, with the exception of the first trial, the results are positive and the heterogeneous version performs better than the homogeneous version and close to the theoretical best. Essentially, the code works by first being told that it has three nodes and that one node can run the program twice as fast as the other two. Instead of breaking the problem into three equal partitions, one for each multiprocessor, as would normally happen, the program divides up the data such that node 0 gets half of the work (since it has half of the combined processors of the cluster) and nodes 1 and 2 each get a quarter of the work. Node 1 and 2 then can work on their parts of the workload without spawning off extra threads and can communicate with each-other and node 0 with MPI. Node 0 first spawns two threads and performs shared-memory communication with Pthreads to communicate between the two threads and MPI to communicate with nodes 1 and 2. In this manner, the program has been able to be sped up via heterogeneous partitioning.

Original	Balanced	Theoretical	Balanced Speedup	Theoretical Speedup
136.978	173.233	182.637333	26.47%	33.33%

Table III.1: redblack3D MFLOPS rates with heterogeneous partitioning using hand-coded MPI and Pthreads. N=300, PE0 has 2 processors and PE1 and PE2 have 1 each.

To analyze what the theoretical best possible speedup might be through heterogeneous decomposition, consider N to be the total amount of data in the

problem. In the homogeneous run, each processor gets $N/3$ data. Node 0 finishes in time T while nodes 1 and 2 finish in time $2 * T$ because they only run half as fast. So two processors are wasted for a time of T. If the data is repartitioned optimally, so that node 0 gets $N/2$ data and nodes 1 and 2 get $N/4$ data each, then the time that each node would take is $T*(N/2)/(N/3) = (3/2)T$, assuming optimal efficiency (the equations for obtaining the new times are described in chapter 5). Although the timings did not indicate optimal efficiency, they approached it and at least gave indications that not only that MPI and Pthreads worked well together, but more importantly, that the repartitioning concept is valid.

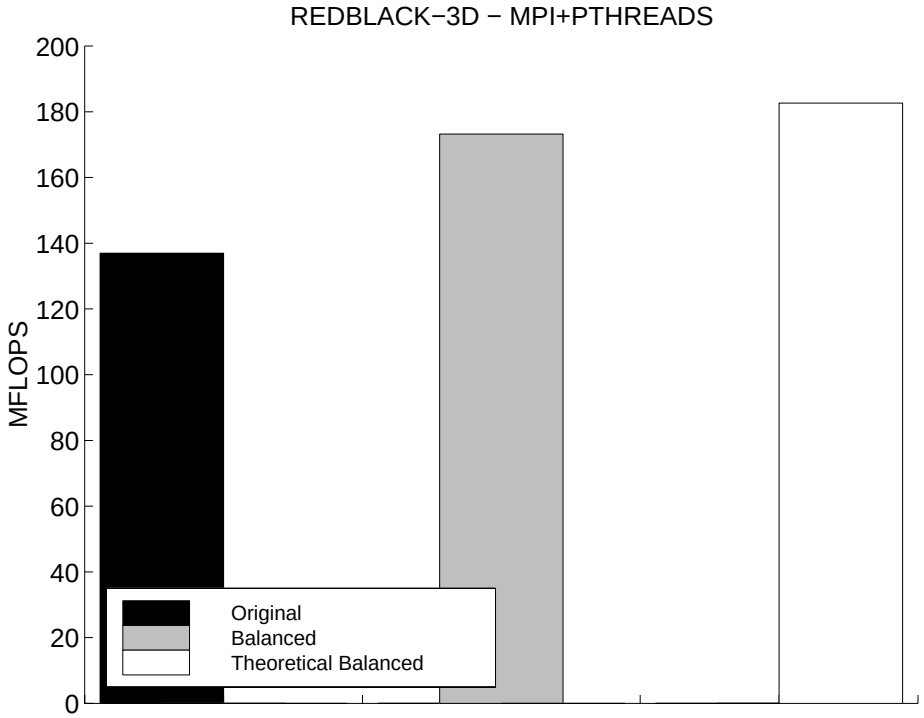


Figure III.2: redblack3D with heterogeneous partitioning using hand-coded MPI and Pthreads. N=300, PE0 has 2 threads and PE1 and PE2 have 1 each.

III.C.2 MPI and OpenMP

Following successful results with MPI and Pthreads, I combined MPI and OpenMP in a single program. The idea, if successful, would make it easier for programmers to create multi-tier programs, since OpenMP is inherently easier to program with than Pthreads, due to its higher-level constructs. I was not certain, however, if thread binding in OpenMP and compatibility with MPI would function properly on the Origin 2000 system. Also, based on the fork-join model that OpenMP is built on (as opposed to the "parked threads" concept that I discussed earlier in relation to Pthreads), I was not sure that the MPI-OpenMP combination would work with all of the different kernels I wanted it to. A *fork-join* model has the threads fork at the beginning of the parallel region declared with compiler directives and join at the end. Thus if the parallel region is called many times, there might be a significant amount of overhead involved in creating and cleaning up threads.

Because of the way some kernels are written, I suspected that there might be problems with this fork-join model. For instance, a tiny amount of computation in a Fortran kernel which is itself inside many nested C++ loops would cause a problem because the machine would be inefficient due to the cost of forks and joins at the beginning of the parallel region inside the Fortran code with each iteration of the C++ loops. An example of this is shown in Figure III.3

Finally, there was the issue of Fortran and C++ OpenMP compatibility since there were Fortran directives recognized separately by each language's compiler. I wrote a small microkernel to test OpenMP scaling as well as C++, Fortran, OpenMP and MPI compatibility on the Origin 2000 as shown in Figure III.4. The microkernel was written to avoid compiler optimization and precomputation as much as possible.

The program runs two heavyweight processes. One process keeps running just the inner loop for every iteration:

```
for(i = 0; i < LONG; i++)
        arr[i] = arr[i-1] * i * 1.0001;
```

```
...
    for (int i = 0; i < 40000; i++)
        times = kernel(x,y,z);
...
double kernel(double x, double y, double z) {

// Every time that kernel() is called, the following
// pragma is called as well.  If kernel() is called
// 40,000 times, as shown in this example, overhead
// of generating threads will be incurred 40,000
// times.
#pragma omp parallel shared(x,y,z)
#pragma omp for schedule(static)
    for(int i = 0; i < MAX_INT; i++) {
        ...
        <mathematical calculations>
        ...
    }
}
```

Figure III.3: OpenMP Fork-Join Example

The other process, for each iteration of the outer loop, runs with a different number of threads, set with OpenMP commands. The program starts at 64 threads and halves the number of threads each time through, all the way down to 1. If the program scales well, the time should double with each iteration, since half the number of processors are working on the problem.

As can be seen from the charts in Figure III.5, figure III.6, and table III.2, for at least up to 8 threads, the kernel scales very well, but for this size problem, does not improve significantly with 16 or 32 threads. Scaling is good but not great, but MPI and OpenMP are shown by the results to interoperate without any problems.

After I ran experiments to determine whether MPI and OpenMP as well as C++ and Fortran OpenMP directives would coexist and function correctly, I set out to obtain timings to see how well OpenMP would parallelize scientific code to see if OpenMP would be close to Pthreads in terms of efficiency, since it already

```
        for (j = 64; j > 0; j=j/2) {
                if (myid != 0) {
                        omp_set_num_threads(j);
                        start = MPI_Wtime();
#pragma omp parallel shared(numthreads)
                        {
                          if( 0 == omp_get_thread_num() )
                            numthreads = omp_get_num_threads();
#pragma omp for schedule(static)
                          for(i = 0; i < LONG; i++)
                                arr[i] = arr[i-1] * i * 1.0001;
                        }
                        finish = MPI_Wtime();
                }
                else {
                        start = MPI_Wtime();
                        numthreads = omp_get_num_threads();
                        for(i = 0; i < LONG; i++)
                                arr[i] = arr[i-1] * i * 1.0001;
                        finish = MPI_Wtime();
                }
                cout << "PE " << myid << "  threads " << numthreads
                     << "  time " << finish - start
                     << "   j " << j << "\n" ;

                MPI_Barrier(MPI_COMM_WORLD);
        }
```

Figure III.4: OpenMP scalability-test code

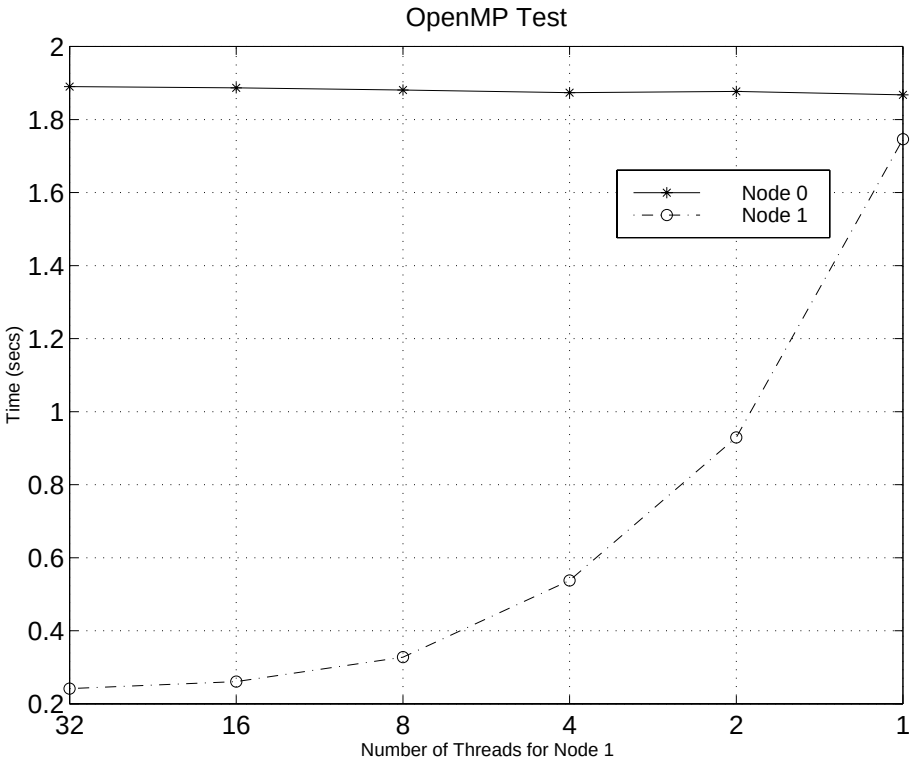


Figure III.5: OpenMP Scaling Test Timings on an SGI Origin2000 with 250 MHz Processors

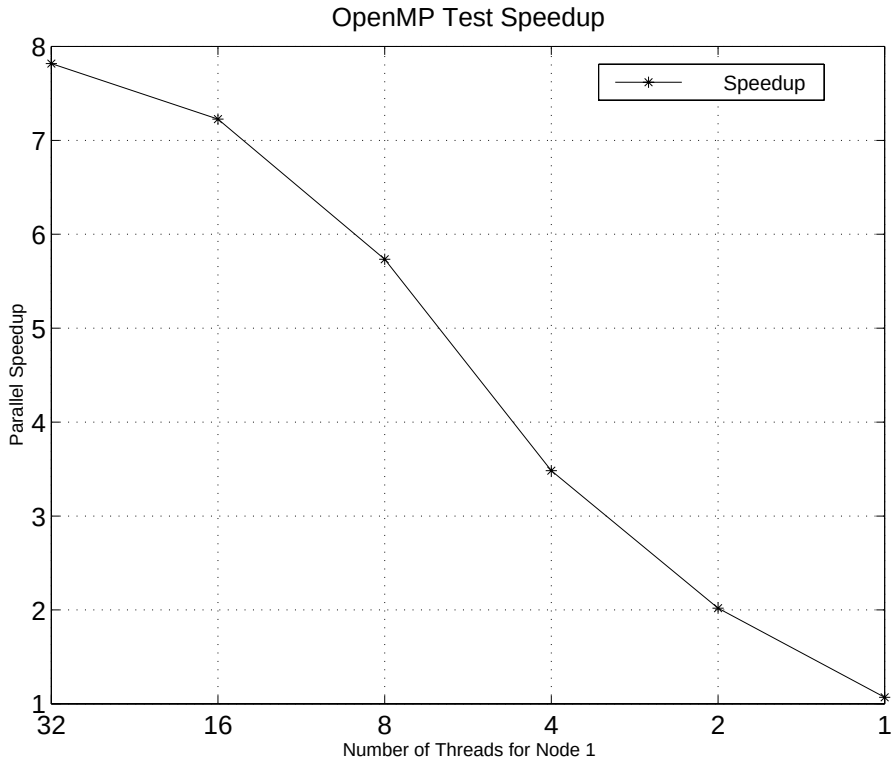


Figure III.6: OpenMP Scaling Test Speedup for an SGI Origin2000 with 250 MHz Processors

Threads	Node 0	Node 1	Speedup
32	1.8800	0.2417	7.8182
16	1.8867	0.2611	7.2263
8	1.8807	0.3280	5.7342
4	1.8733	0.5378	3.4833
2	1.8768	0.9297	2.0186
1	1.8677	1.7462	1.0696

Table III.2: Table of OpenMP Scaling Test Speedup for an SGI Origin2000 with 250 MHz Processors

had an advantage in ease of programming.

III.C.3 KeLP and OpenMP

Finally, because MPI (one of the component technologies that KeLP is built on) functioned properly with OpenMP, I assumed that KeLP would function properly with OpenMP. However, prior to adding an API to KeLP directly, I decided to test both KeLP and OpenMP's compatibility and also the efficiency of programs built using both technologies. I went back to rb3D this time, but instead of starting with a hand-coded MPI and Pthreads version, I began with a KeLP [2] version, then added OpenMP directives and code to have the two interact (which would be later replaced by calls directly to the new Sputnik API if the experiment was successful). After seeing that the program compiled and appeared to run successfully, I had all of the elements in place to write the Sputnik API and begin gathering data on its effectiveness.

[2]KeLP Web Page: <http://www-cse.ucsd.edu/groups/hpcl/scg/kelp/>

Chapter IV

The Sputnik Model and Theory of Operation

IV.A Introduction

Using the experience gained from experiments with hand-modifying various combinations of MPI, KeLP, OpenMP and Pthreads programs, I extended the API and functionality of KeLP in a new set of routines called Sputnik. These routines that I implemented perform two steps that work in tandem to achieve efficiency on heterogeneous clusters.

Although the Sputnik Model allows for any sort of shared-memory multiprocessor and any sort of optimizations to be done, in theory, the Sputnik API has been written with a specific focus. The API has been written with two of the many possible optimizations, that are validated as good optimizations for the redblack3D benchmark in the next chapter. The order of events for the Sputnik Model, which the API based on, is:

1. *ClusterDiscovery*: Runs the kernel of the program repeatedly on each separate node to determine the timings and relative performance, changing program parameters over several runs to determine the configuration that

achieves optimal performance.

2. *ClusterOptimization*: Using the parameters which the program estimates to be optimal from ClusterDiscovery, decomposes the data non-uniformly based on the relative powers of the nodes.

3. Runs the program one last time using the optimizations and decompositions from ClusterDiscovery and ClusterOptimization.

IV.B Sputnik Model

IV.B.1 ClusterDiscovery

The ClusterDiscovery performs an estimation. It searches the parameter space, somewhat intelligently, for the available optimizations, and seeks a performance gain in the program it is optimizing. The ClusterDiscovery works transparently to the user. Since one of the goals of the Sputnik Model is to allow the user to program as if the cluster is heterogeneous, the ClusterDiscovery runs through the possible optimizations automatically and finds the best optimizations and the timings for those runs using the optimizations. The kernel runs inside a kind of "shell" so that Sputnik has access to running it whenever it needs to. Not only does the ClusterDiscovery save the user from manually searching for all the optimal configuration parameters, but there is no firm limit on the amount of permutations that can be searched since the search all happens transparently inside its shell.

Optimizations could include adjusting the number of OpenMP threads (as is done in the Sputnik API), doing cache tiling, sending vectorizable code to a vector computer in the cluster and parallelizable code to an MPP in the cluster; and a huge number of other possible variations in the entire parameter space of possible optimizations for scientific code.

This shell is run separately on each separate node in the cluster so that each node is optimized individually with a distinct parameter set and timing results are returned to the shell for those optimizations.

One does not need to know anything about the characteristics of each node in the cluster, which may or may not even be multiprocessors, prior to running a program written using the Sputnik Model.

IV.B.2 ClusterOptimizer

The ClusterOptimizer uses the optimizations found in the ClusterDiscovery stage and the best timings of each node to decompose the computation of the problem according to the performance of each node in the cluster. A node discovered to have better performance than other nodes will therefore work on a larger chunk of the problem. Depending on the size of the problem as a whole, the cache sizes, and the amount of communication taking place, there are a variety of different decomposition schemes available, which are described below. The important thing, however, is to make sure that which ever decomposition scheme is used, computation must be balanced out so that each node finishes at the same time. Finally, one must make sure that after decomposition, communication does not overwhelm computation. To the degree that the original problem does not have this issue and we insist that no node is slower than half the speed of the fastest node, there should not necessarily be any inherent restrictions on which type of decomposing to use to partition the data for the heterogeneous cluster.

IV.B.3 Running

Given the optimal parameters and partitioning, we know enough to run the program on a heterogeneous cluster and can expect it to utilize the cluster and perform significantly better. Because results of ClusterDiscovery are saved on disk, future runs of the program on the same cluster will not have to “re-discover”

the cluster each time and can simply run with the optimal settings.

IV.C Sputnik API

IV.C.1 Goals

The Sputnik API is based on the Sputnik Model. It implements a specific subset of the ideas generalized in the Model.

1. *ClusterDiscovery*: Runs the kernel of the program repeatedly on each separate node to determine the timings and relative performance, varying the threads per node given to determine the optimal number of threads per node.

2. *ClusterOptimization*: Using the parameters which the program estimates to be optimal from ClusterDiscovery, it decomposes the data non-uniformly based on the relative powers of the nodes.

3. Runs the program one last time using the optimizations and decompositions from ClusterDiscovery and ClusterOptimization.

The first optimization that Sputnik performs is the determination of the optimal number of OpenMP threads per node to run with. The optimal number of threads might be equal to the number of processors in the node but may be less if the problem is not large enough to warrant the overhead and costs of shared-memory communication for that many threads. More threads would not speed up communication and may in fact slow it down because two or more threads would be competing for one processor's time and memory.

The second optimization that Sputnik performs is decomposition. One of the appealing aspects of using KeLP, aside from the fact that it is based on MPI, which has a standardized interface across all major parallel platforms, is that it supports *block decomposition*.

IV.D Decomposition

IV.D.1 Block Decomposition

Block decomposition allows for manipulating certain types of data in a much easier way than with standard C++ datatypes. Also, rather than describing data in terms of C++ arrays and having to program complex MPI communications, in KeLP, one can describe the domain of the data, called a `Grid` and multidimensional blocks within the domain called `Region`s. Rather than forcing the application programmer to write long sequences of loops to pass ghost cells (boundary conditions between the blocks) between processors, KeLP handles this automatically with built-in functions to expand and shrink any given `Region`. Further, a `Region` can be intersected with another Region and since it is merely a subset of the overall `Grid`, it can overlap with another `Region` and cover a domain which may as a whole be very irregular, with neat, individual blocks.

Block decomposition facilitates the implementation of features including tiling for cache and repartitioning. The Sputnik *Model* is different from the Sputnik *API*. In the Sputnik Model, I discuss how an API like KeLP can discover resources and act appropriately to refine the program to run heterogeneously. In my API, I have made specific choices and assumptions about what to implement. Although, as I have mentioned, many possible optimizations could have been made, I specifically chose repartitioning and thread adjustment as methods for optimizing for heterogeneous clusters.

IV.D.2 Heterogeneous Partitioning

KeLP provides a mechanism for doing automated decomposition of a `Grid` object into `Domain` objects and further into `FloorPlan` objects, which include assignments of each `Region` to processors. This mechanism is contained in a library called *DOCK* (DecOmposition Classes for KeLP). Among other uses, DOCK will take a `Grid` and partition it into equal size `Region`s, with slight tolerance and

variation when the number of `Region`s to divide the `Grid` into does not evenly divide the amount of columns or rows index to be partitioned. Since a `Region` or `Grid` can be multi-dimensional, DOCK can partition in multiple dimensions as well. Roughly, the one-dimensional partitioning of a two-dimensional `Grid` into three two-dimensional `Region`s might look like the partitioning in figure IV.1.

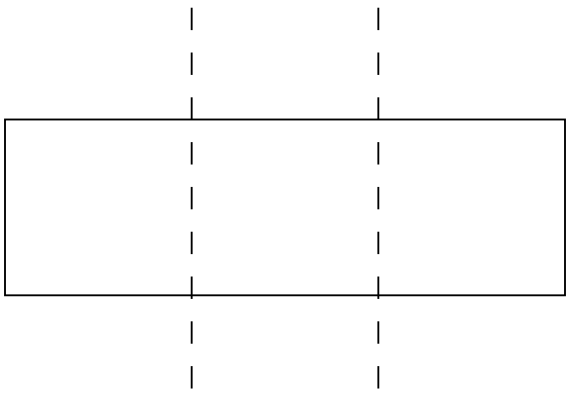


Figure IV.1: Two-dimensional dataset partitioned into three equal two-dimensional blocks.

Thus, each block, or `Region`, would have one-third of the dataset, which would in turn get assigned to a cluster (where each node gets only one MPI process). However, a question this thesis addresses is: What happens if each node is not equal in processing power? What if instead of the processing power of the cluster being as evenly divided as the data is in figure IV.1, such that node 0 is twice as powerful as the other two nodes and so therefore can run in 10 seconds whatever nodes 1 or 2 can run in 20 seconds. In this case, the partitioning of the dataset should look like the one in figure IV.2.

By modifying the DOCK library so that the domain is partitioned non-uniformly, according to how well each node really performs, rather than a uniform partitioning, the program can be run on the cluster more optimally.

In the previous chapter, I discussed this partitioning scheme for use with the redblack3D version that runs with hand-tuned MPI and Pthreads and described

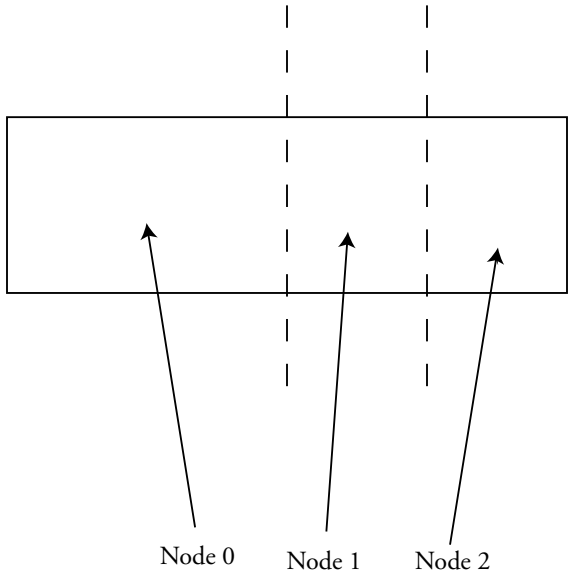


Figure IV.2: Two-dimensional dataset partitioned so that node 0 gets twice as much data to work with as either node 1 or node 2.

how the relative power of a node can be determined by comparing the inverse of its time with the sum of the inverse of the times, forming a ratio. This ratio is multiplied by the total amount of work available to determine the size of the block to give to a particular node.

IV.E API Design

The API is designed so that the `main()` routine of a program is moved, mostly, to a user-defined routine called `SputnikMain()`. The real `main()` does initialization and calls a routine called `SputnikGo()`. `SputnikGo()` acts as a kind of "shell" that calls `SputnikMain()` over and over to determine the optimal number of threads per node, and make the final run with the optimized configuration. The repartitioning, one of the primary features of the Sputnik API is a modification of the distribution functions in the DOCK library. Although no new functions are added, the distribution functions are mostly rewritten to support non-uniform

partitioning.

IV.F Assumptions and Limitations

There are assumptions this API makes. First, it assumes that no node is more than twice as fast as another node. This assumption also helps to ensure that communication time does not overwhelm a particular node because it is doing so much less computation than another node. Second, because the API is running the *entire* kernel on each separate node, it assumes that the speed of the node will not change when the node is given only a *portion* of the entire computational domain to run, as is done after the repartitioning, for the final run. Finally, the repartitioning that Sputnik does is only one-dimensional. Although DOCK (and therefore KeLP) support multi-dimensional decomposition, for simplicity, Sputnik does not.

One reason to support multi-dimensional decomposition is if Sputnik will be running on a cluster with a huge number of nodes so that memory and cache tiling could be built into the decomposition and message-passing communication could be made automatically more efficient. Since the initial release of the API has focused on much smaller clusters with 2-4 nodes, I assumed that one-dimensional decomposition was adequate and the possible downsides would be negligible.

The Sputnik API also requires that the application is written in C++, at least as a wrapper, though the kernel(s) of the program may be written in either C, C++, or Fortran and linked in. Finally, Sputnik depends on the fact that the cluster has a thread-safe implementation of MPI installed as well as OpenMP for both Fortran and C/C++ and all technical requirements for both MPI and OpenMP programs to run are adhered to.

The validation of these results and of the model appear in the next chapter.

Chapter V

Validation

V.A Introduction

Redblack3D was the program I chose to run on a pair of SGI Origin2000 supercomputers at the National Center for Supercomputing Applications (NCSA) to determine the success of the performances aspect of the Sputnik API [48][47]. Whether it succeeds also in its goal of being able to allow the user to program a heterogeneous cluster as if it is homogeneous is much harder to measure, although there are not very many changes that have to be made, and I estimate that someone familiar with OpenMP could make the modifications to a program that already runs in KeLP in an hour. As already discovered in several papers, however, some scientific codes optimizes better than others [21][44][45].

V.A.1 Red-Black 3D

Redblack3D it is a scientific program written using Sputnik, which itself comprised of MPI, KeLP and OpenMP. It is mostly in C++, except for the kernel of the program, which is written in Fortran and linked in with the rest of the code. The OpenMP directives are in the Fortran part of the code. The program itself solves Poisson's equation (a differential equation) using the Gauss-Seidel method.

Each run was done with one MPI process per Origin2000 and varying numbers of OpenMP threads within each system. The program alternates between running a kernel on the "black" points and the "red" points, which are arranged in a three-dimensional grid. The reason for this is that redblack3D operates by doing a 7-point stencil calculation (right, left, front, back, top, bottom, and itself) and needs to make sure that the values next to it don't change in a given iteration. So the grid alternates with every-other point being a red point and all the others being black with no red immediately adjacent to a black point (though orthogonal is fine). The entire source code for redblack3D and the kernel, both modified with the Sputnik API, are in Appendices B and C.

V.B Hardware

The Origin2000 is somewhat different than the typical multiprocessor which inspired this thesis because it uses distributed shared memory. Each node on the Origin2000 consists of two processors which have locally shared memory. The nodes are all connected together in a complex structure to achieve 128 to 256 processors per system. A diagram of this is shown in figure II.3. This large system, since it has distributed shared memory, can be used to simulate a multiprocessor, although one could argue that an Origin2000 itself is really a collection of tightly coupled SMPs. Therefore, since in these test cases, I used a large part of the Origin2000 and a node is only a small part of the system, in this results chapter, I will refer specifically to a "system" where I have referred interchangeably to either a *node* or multiprocessor in previous chapters. Likewise, since before, I ran with one MPI process per multiprocessor node, here, I will run with one MPI process per Origin2000 system. Unlike an SMP, this distributed shared memory system is not an UMA machine. Instead, it is a NUMA derivative called *cache-coherent, non-uniform memory access* or *ccNUMA*.

I obtained my results by performing experiments on the two machines,

balder and *aegir*, shown in Table V.1.

	balder	*aegir*
Processor Type and Clock Speed	250 MIPS R10000	
Cycle Time	4.0 ns	
Processor Peak Performance	500 MFLOPS	
L1 Cache Size	32 KB	
L2 Cache Size	4 MB	
Operating System	IRIX 6.5	
Compilers and Linkers	Native SGI Fortran and C++	
Processors	256	128
Main Memory	128 GB	64 GB
Peak Theoretical Performance	128 GFLOPS	64 GFLOPS

Table V.1: Specifications for the two Origin2000 machines, *balder* and *aegir*.

Although I experimented with a variety of environment variables in many possible combinations, I found the optimal settings for "_DSM_MIGRATION" to be "ALL_ON" and "_DSM_PLACEMENT" to be "ROUND_ROBIN." The system software configurations that I used, and their versions, are shown in table V.2. The two Origin2000 machines were connected by an SGI "Gigabyte System Network (GSN)" interconnect that support a maximum bandwidth 800 MB per second and have a theoretical latency of less than 30 microseconds. Experimental results showed that the actual latency might be much closer to 140 microseconds and the bandwidth less than 100 MB per second.

	Version	Special Flags
Operating System	IRIX 6.5	
MIPSpro f77	7.3.1m	-mp -O3 -mips4 -r10000 -64
MIPSpro CC	7.3.1m	-mp -lmpi -lm -lftn -lcomplex -O3 -r10000 -64
KeLP	1.3a	

Table V.2: Software configurations

V.C Predicted Results

Generically, the optimal speedup can be computed from the following equations, where T is total time. For all cases, both with heterogeneous and homogeneous partitioning, the total time for the program is only as fast as the slowest node:

$$T_{program} = T_{slowestnode} \tag{V.1}$$

Thus, for a cluster, if one node runs faster than the rest, the fastest node is wasting time while it waits for the slower nodes to finish. Thus the wasted time can be computed as follows, where N is the total number of nodes and the times for the nodes are ranked from 0 to $N-1$:

$$T_i = \text{TimeOnNode}i \tag{V.2}$$

$$T_{wasted} = MAX(T_i) - \sum_{i=0}^{N-1} \frac{T_i}{N} \tag{V.3}$$

Therefore, the solution is to repartition the data. After optimally repartitioning, the program will run in a time in-between that of the faster and slower nodes:

$$T_{optimal} = T_{i,orig} * \frac{\text{newamountofdatafornode}i}{\text{originalamountofdatafornode}i} \approx \sum_{i=0}^{N-1} \frac{T_i}{N} \tag{V.4}$$

Therefore, the speedup will be:

$$Speedup = \frac{T_{wasted}}{T_{optimal}} = \frac{MAX(T_i)}{T_{optimal}} - 1 \tag{V.5}$$

This speedup is based on a repartitioning of the dataset. Where in a homogeneously partitioned run, each node would be assigned $1/N$ work, in a heterogeneously partitioned run, the work assigned is a bit more complex. If the node originally ran in time $T_{i,orig}$ then first we want to find out what the speed of this

node is relative to all the others and the total. We can do this by assigning a power P to the node equal to the inverse of the time. We can use this then to produce a power ratio R.

$$P_i = \frac{\sum_{j=0}^{N-1} T_j}{T_{i,orig}} \tag{V.6}$$

$$P_{totalcluster} = \sum_{k=0}^{N-1} P_k = \sum_{k=0}^{N-1} \frac{\sum_{j=0}^{N-1} T_j}{T_{k,orig}} \tag{V.7}$$

$$R_i = \frac{P_i}{P_{totalcluster}} = \frac{\sum_{j=0}^{N-1} T_j}{T_{i,orig}} / \sum_{k=0}^{N-1} \frac{\sum_{j=0}^{N-1} T_j}{T_{k,orig}} \tag{V.8}$$

The goal of the software is to assign a percentage of the total work of the problem to a node such that the node has the same percentage of work to be done relative to the total amount of work as the power of the node is relative to the total power of the cluster. Thus,

$$R = \frac{T_{i,orig}}{T_{total,orig}} = \frac{work_i}{work_{total}} \tag{V.9}$$

and so finally the amount of work we give to each node is:

$$work_i = R_i * work_{total}. \tag{V.10}$$

In the case of round-off issues, some work chunks will have 1 added to them to make sure all data is accounted for.

Plugging this back into our original equation for predicting the new time,

$$T_{optimal} = T_{i,orig} * \frac{\text{newamountofdatafornode}i}{\text{originalamountofdatafornode}i} \tag{V.11}$$

$$= T_{i,orig} * \frac{work_{i,new}}{work_{i,orig}} \tag{V.12}$$

$$= T_{i,orig} * \frac{R_i * work_{total}}{work_{i,orig}} \tag{V.13}$$

$$= T_{i,orig} * \frac{\frac{P_i}{P_{total}} * work_{total}}{work_{i,orig}} \tag{V.14}$$

$$= T_{i,orig} * \frac{\frac{\sum_{j=0}^{N-1} T_j}{T_{i,orig}}}{\sum_{k=0}^{N-1} \frac{\sum_{j=0}^{N-1} T_j}{T_{k,orig}}} * \frac{work_{total}}{work_{i,orig}} \quad \text{(V.15)}$$

$$= T_{i,orig} * \frac{work_{total}}{work_{i,orig}} * \frac{\sum_{j=0}^{N-1} T_j}{T_{i,orig} * \sum_{k=0}^{N-1} \frac{\sum_{j=0}^{N-1} T_j}{T_{k,orig}}} \quad \text{(V.16)}$$

$$= \frac{work_{total}}{work_{i,orig}} * \frac{\sum_{j=0}^{N-1} T_j}{\sum_{k=0}^{N-1} \frac{\sum_{j=0}^{N-1} T_j}{T_{k,orig}}} \quad \text{(V.17)}$$

V.D Experiments

The purpose of each of the experiments that I ran was to determine how close to the optimal time that the run could come by repartitioning with the Sputnik library, regardless of the numbers of threads actually used or the size of the problem. The experiment is designed to establish, artificially, various levels of heterogeneity, in order to detect any sensitivity in Sputnik. The results showed that Sputnik was insensitive to this parameter and so the degree of heterogeneity, as long as no node is less than half as fast as any other node, is not relevant.

As I have said, Sputnik can find the optimal number of OpenMP threads per system to use. Alternatively, these can be manually and individually set. Principally, I manually set the amount of OpenMP threads per system so as to focus more on finding the right partition than finding the right number of threads per system. I ran with many different configurations of threads per system, however, so in effect, I was able to manually try to find the optimal number of threads per system. The reason for this is that using a system with 128-256 threads per system, one might have to do 30 or more runs to find the optimal number of threads per system and the time on the Origin2000's was not available.

Additionally, do to scaling issues, I did not use the full number of processors on each Origin2000. By manually setting the number of threads, I created a "virtual cluster." The virtual cluster had the effect of simulating a heterogeneous cluster since the full Origin2000 cluster could not be used and no other "commodity

clusters," as the Sputnik API was designed for, were available.

To that end, I ran with several different values. First, I fixed the size of the problem to N=761 (761x761x761 unknowns) and the number of threads on *balder* to 32. I then ran redblack3D once for each different number of threads for *aegir* that I wanted to test with. For *aegir*, I started with 16 threads and ran again with 20, 24, 28 and 32. The reason that I started with 16 is that I made the decision ahead of time that if one system was less than half as powerful as the other that it probably would not even be worth using the slower system. Therefore, I scaled my problem from one-half of the fixed number of threads on *balder* up to the same number of threads on *aegir*.

After doing the experiments with 32 threads on *balder*, I ran with the identical problem size with 48 threads on *balder*, this time scaling from 24 to 30, 36, 42 and 48. Finally, I ran just a few very large problem sizes, with up to 128 threads on *balder* and 96 on *aegir*, as I will discuss below.

In the tables and graphs that follow, I include many different types of times. Although one can compare the original and new *total* times (communication plus computation) to get the most realistic "real-world" times, comparing the computation times alone shows the results better. The reason for this, as I will show in a few limited runs, is that when times are observed when communication is measured for homogeneously partitioned runs, the times for both nodes will be nearly identical. The reason for this is that they need to synchronize at various points and so both maintain a sort of lock-step with each other at certain barriers. Even though the times are similar, however, they are both very high. They are clearly demonstrating the fact that a program can run only as fast as its slowest node. It is more interesting and informational, however, to see the times for exclusively computation.

V.E Results of redblack3D

V.E.1 Up to 32 threads per system

Threads	Original Compute		New Total		New Compute	
aegir	*balder*	*aegir*	*balder*	*aegir*	*balder*	*aegir*
16	47.7	87.7	66.415	65.7	65	60.4183
20	48.4	74.4	63.0848	63.8	61.3	58.0324
24	51	62.7	58.4	59.5019	56	55.4033
28	50	55.6	54.409	53.8	50.3	51.4442
32	51.2	48.7	51.6	52.5732	48	50.5865

Table V.3: Complete redblack3D timings with 32 threads on *balder* and varying numbers of threads on *aegir*

Threads		New Compute		Predicted	Speedup	Theoretical
balder	*aegir*	*balder*	*aegir*			Speedup
32	16	65	60.4138	61.7916	1.3492	1.4193
32	20	61.3	58.0324	58.6476	1.2137	1.2686
32	24	56	55.4033	56.2480	1.1196	1.1147
32	28	50.3	51.4442	52.6515	1.0808	1.056
32	32	48	50.5865	49.9187	1.012	1.0257

Table V.4: Speedup and predicted timings for redblack3D with 32 threads on *balder* and varying numbers of threads on *aegir*

Figure V.1, table V.3, and table V.4 show the results for the runs of redblack3D with 32 threads on *balder* and 16 to 32 threads on *aegir*. The important numbers to compare are the *slowest* of the original times for computation to the slowest of the new times for computation. This is because, as mentioned before, the program only runs as fast as the slowest system. Therefore the time for the slowest system is also the time for the whole program. The goal of Sputnik is to make the program run faster. A side benefit of this is that it also increases machine utilization by not having a system or systems remain idle while waiting for the slower system or systems to catch up. In table V.3, we can see the original,

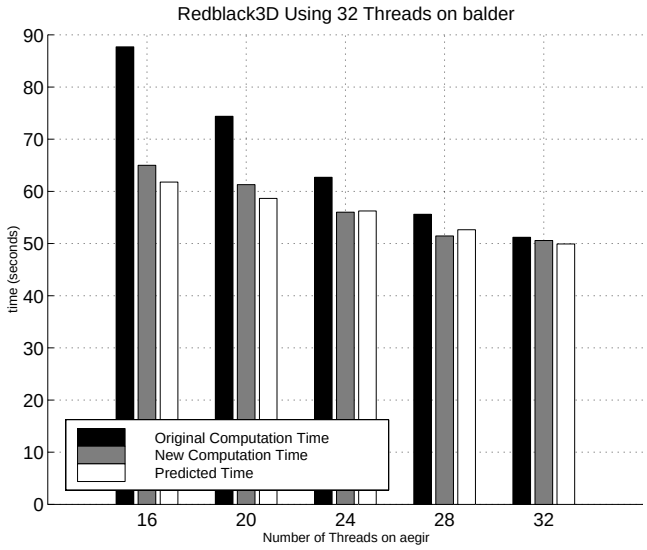


Figure V.1: Important redblack3D timings with 32 threads on *balder* and varying numbers of threads on *aegir*

unmodified run with 16 threads on *aegir* causing *balder* to remain idle for up to 40 seconds while waiting for *aegir* to catch up.

As can be seen clearly from figure V.2, Sputnik does provide a speedup over the version of the program without the heterogeneous Sputnik partitioning. As expected, the speedup tapers off as the number of threads per system becomes closer together on the pair of Origin2000's. Not only does Sputnik provide, but as shown, it demonstrates improved system utilization because one system does not remain idle for nearly as long as it originally had. At its best, Sputnik shows a 34.9% speedup when 32 threads are used on *balder* and 16 threads are used on aegir. This speedup is good, though still 7% less than the theoretical best for the thread configuration with 32 threads on *balder* and 16 threads on *aegir*.

The "New Compute" timings for each system should be close to identical. As the timings from table V.3 indicate, the speeds are not perfectly identical. The reasons for this are not completely clear, but there are a number of possible explanations. First, speed on the Origin2000 depends highly on thread scheduling

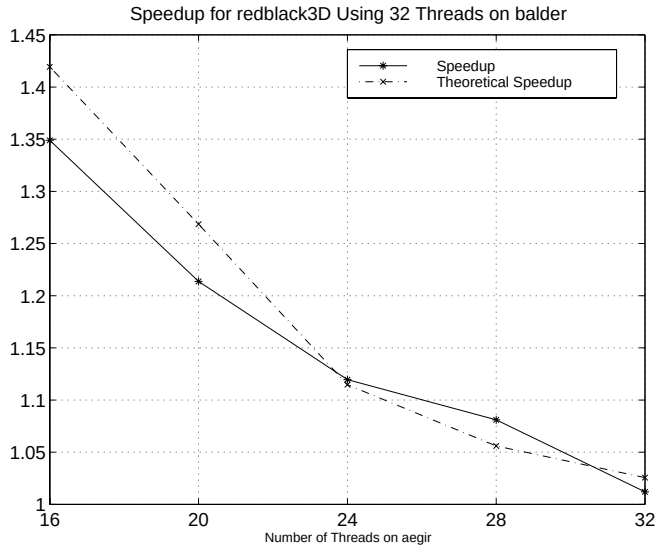


Figure V.2: Speedup for redblack3D with 32 threads on *balder* and varying numbers of threads on *aegir*, using the Sputnik library

and memory placement, both of which seem to be massive issues affecting performance and can vary widely even within the run of a program, as in this case. Since memory is stored throughout the entire machine within the two-processor nodes, a poor distribution could certainly affect the timings and cause the kind of variation that we see in table V.3. Because these timings were done in a dedicated environment, competition for processor, memory, or network bandwidth is not an issue.

Based on the equations presented above, we can predict, from the original timings and the number of threads per system that we are using, the theoretical or predicted time for a re-partitioned and balanced run of the program. At its worst, the runs with a maximum of 32 threads per system are 5.19% worse than the predicted results. At best, the actual runs are 2.29% better than the predicted results. Again this variance is presumably due to the same conditions that cause a discrepancy between the timings of the nodes after their workloads have been re-balanced.

V.E.2 Up to 48 threads per system

Threads	Original Compute		New Total		New Compute	
aegir	*balder*	*aegir*	*balder*	*aegir*	*balder*	*aegir*
24	37.6	62.9	50.6	49.8617	43.8	46.3449
30	37.6	50.1	45.498	45.5	43.4	42.687
36	36.1	43.5	42.2992	42.3	37.8	39.675
42	36.5	38.4	40.5622	40.7	36.8	35.4458
48	37.4	34.3	41.4	42.3218	33.3	36.2054

Table V.5: Complete redblack3D timings with 48 threads on *balder* and varying numbers of threads on *aegir*

Threads		New Compute		Predicted	Speedup	Theoretical
balder	*aegir*	*balder*	*aegir*			Speedup
48	24	43.8	46.3449	47.0655	1.3572	1.3364
48	30	43.4	42.687	42.9592	1.1544	1.1662
48	36	37.8	39.675	39.4560	1.0964	1.1025
48	42	36.8	35.4458	37.4259	1.0435	1.0260
48	48	33.3	36.2054	35.7830	1.0330	1.0452

Table V.6: Speedup and predicted timings for redblack3D with 48 threads on *balder* and varying numbers of threads on *aegir*

As with the runs with a maximum of 32 threads per system, runs with a maximum of 48 threads per system also scaled well. In this case, I varied the number of threads on aegir from 24 up to 48. Table V.5 shows the complete list of timings, as does figure V.3. The important timings, showing only the slowest system from each run, are indicated in figure V.4. The speedup, which is very good is shown in figure V.5 and table V.6. In this case, the results for running with 48 threads on *balder* and 24 threads on *aegir*, are even slightly better than the case with 32 and 16, showing 35.7% speedup.

As with the runs with a 32-thread maximum per system, these runs also differed from the predicted runs somewhat, but to a lesser extent. In the cases when I ran with 48 threads per system, at best, Sputnik produced results 1.67%

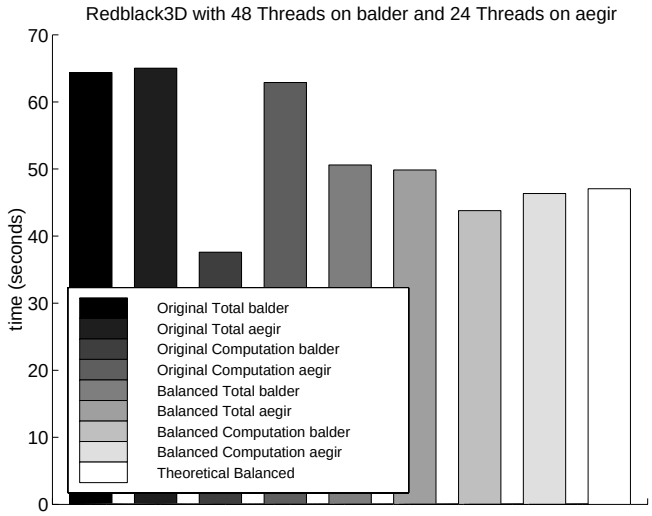


Figure V.3: Complete timings for redblack3D with 48 threads on *balder* and varying numbers of threads on *aegir*, using the Sputnik library

better than predicted and at worst, it produced results 1.03% worse than predicted.

V.E.3 Large numbers of threads per system

With the very large numbers of threads per Origin2000 system, I increased the problem size because I was not finding good scaling with N=761. Therefore for the case with 64 and 32 threads, I had N=949. For 128/64 and 128/96 threads, I used N=1163.

As seen from the timings in table V.7 and especially from the speedups in table V.8, Sputnik performed well with large numbers of threads per system as well, showing more than 34% improvement after repartitioning for either the 64/32 threads case or the 128/64 threads case. As expected, when the numbers of threads per node narrows (and as the number of threads in total grows), the speedup gains decline, indicated by only a 6.77% improvement with the 128/96 threads case.

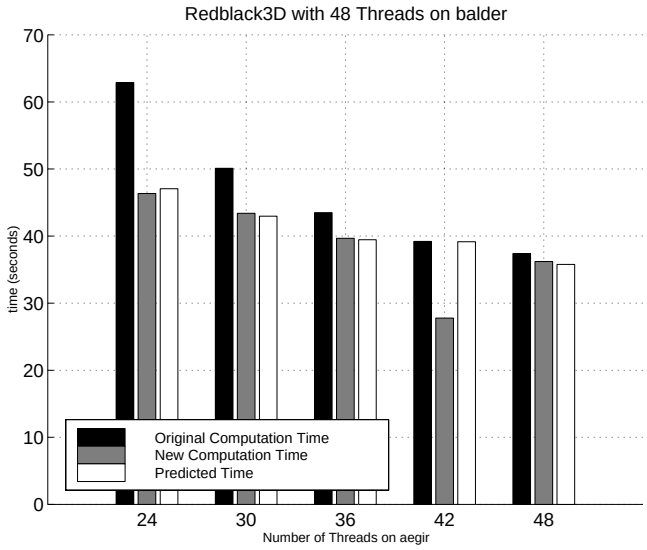


Figure V.4: Important redblack3D timings with 48 threads on *balder* and varying numbers of threads on *aegir*, using the Sputnik library

Threads		Original Compute		New Total		New Compute	
balder	*aegir*	*balder*	*aegir*	*balder*	*aegir*	*balder*	*aegir*
62	32	57.1	99.3	74.6	74.6085	71.5	71.8537
128	64	59.6	108	83.2697	83.7	80.3	75.5464
128	96	65.6	73.5	72.9498	72.7	65.5	68.841

Table V.7: Complete redblack3D timings for large numbers of threads per system

V.E.4 Anomalies

There were some strange things that occurred in the course of gathering results for redblack3D modified with the KeLP/OpenMP/Sputnik API libraries. All of them appear to stem from anomalies either with OpenMP in general, or perhaps the specific OpenMP implementation on the Origin2000 machines. Specifically, I found that MPI processes appear to run significantly faster than OpenMP threads. For example, when I ran with one MPI process with between 2 and 64 threads spawned by that MPI process, it ran significantly slower than if I ran with between 2 and 64 MPI processes with 1 OpenMP thread per process. Despite

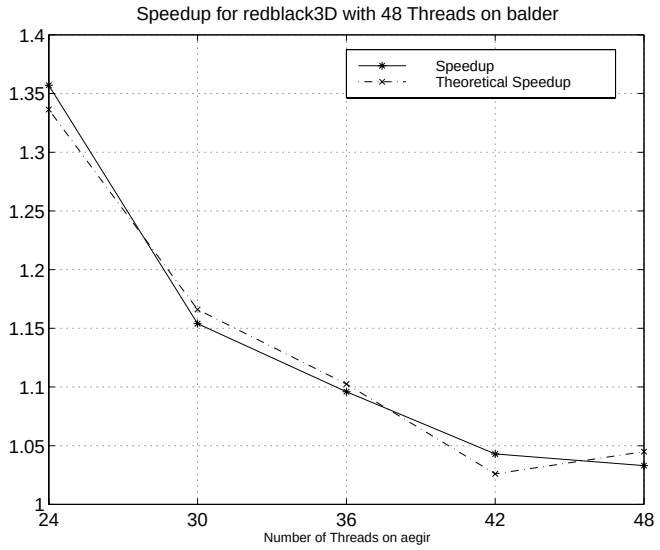


Figure V.5: Speedup for redblack3D with 48 threads on *balder* and varying numbers of threads on *aegir*, using the Sputnik library

Threads		New Compute		Predicted	Speedup	Theoretical
balder	*aegir*	*balder*	*aegir*			Speedup
62	32	71.5	71.8537	72.5068	1.3820	1.3695
128	64	80.3	75.5464	76.8115	1.345	1.4060
128	96	65.5	68.841	69.3257	1.0677	1.0602

Table V.8: Speedup and predicted timings for redblack3D with large numbers of threads per system

speaking with the NCSA Consultants, who in turn contacted SGI engineers, as well as performing a variety of experiments myself, the precise cause of this was never solved. It was speculated that this too had something to do with memory distribution as well as thread distribution (having all threads spawned on the same processor). On the Origin 2000, threads and memory can be distributed throughout the system. There may be processors spawned on one part and memory placed on another, using OpenMP. Despite memory and thread migration and round-robin memory distribution, having all the OpenMP threads realize good memory access speed does not seem to be trivial.

Due to the fact that the program *did improve* with greater numbers of threads, the thread distribution idea could be discounted. What was left was memory distribution and this was never solved.

I decided in the end, though, that since Sputnik demonstrated the results of my thesis — that performance results could be analyzed and that action could be taken, in the case of Sputnik, adjusting the number of threads per system and re-partitioning the dataset according to "processing power" — that my thesis had been proven. The Sputnik API is currently designed to work with a commodity cluster with a good MPI and OpenMP implementation. It will take more work on other vendors' systems with other vendors' implementations of OpenMP to determine exactly what the cause of the problems with OpenMP on the Origin2000 are.

Another problem, presumably related to the OpenMP issues as well, had to do with scaling. First, it turned out that the loop that was most obviously the one to put the OpenMP directives around did not produce any parallel speedup at all. Second, it turned out that only when the program's built-in cache tiling mechanism was disabled by using two specific options (`-si 1 -sj 1`), did the program produce any scaling as well. (And then, as I mentioned before, it performed several times worse than KeLP (which uses MPI exclusively) alone.

Again, since the repartitioning increased the utilization of the cluster and speed of the program, I did not feel that this affected the validity of my results. I am confident that experiments on a true commodity cluster of multiprocessors, as Sputnik was designed for, will resolve the OpenMP scaling and speedup issues.

Chapter VI

Conclusions and Future Work

The results that Sputnik produced with the application study of red-black3D indicate strongly that as part of the ClusterOptimization step discussed in chapter 1, repartitioning the load to balance out the time that each Origin2000 system spends running so that both nodes finish at the same time, works.

The speedup over running unoptimized with uniform partitioning, though not completely linear, is good, and works well for medium-sized problems to very large ones. The most dramatic results came with the example of 48 threads on *balder* and 24 threads on aegir where the repartitioning revealed a speedup of 35.7%. This speedup is actually 2.1% better than the equations predict for the theoretical results. This is shown despite OpenMP anomalies encountered. In fact, all of the results show that Sputnik gives a speedup within 5.2% of the theoretical optimum and the majority of the results are within 2%. Some, in fact, are up to 2.3% better than the theoretical optimum. Figure V.5 shows graphically really how close Sputnik comes to perfect speedup with the available optimizations.

An application written with KeLP can be converted to Sputnik very easily as long as a good OpenMP implementation exists and the kernel that one is trying to parallelize can in fact be taken and optimized by OpenMP well. As noted by other researchers, this is not always possible [21][44][45]. This makes code for heterogeneous clusters almost as easy to program as homogeneous clusters by using

Sputnik instead of KeLP, which was one of my primary goals.

Following the Sputnik Model, the Sputnik API library could certainly be adapted to work with different component technologies. For instance, instead of KeLP1 and OpenMP, it could be built on top of the one of the already-existing multi-tier API's described in my related work section. The intent would be to continue to make developing Sputnik-based scientific code supporting heterogeneous clusters of multiprocessors even easier than Sputnik currently provides, while still achieving at least the speedup that was demonstrated on the Origin2000's at NCSA.

Regardless of the component technologies used, however, the idea of a ClusterDiscovery and ClusterOptimization component appears to work. This can certainly be extended in the future to function well with problems that may benefit not only from a partitioning or balancing of the problem, but possibly from other optimizations including cache tiling or focusing specific sections of the program on specific machines that have unique characteristics from which the sections would benefit.

The idea could certainly also be adapted to work in a dynamic environment as well, where instead of sampling just once, at the beginning, testing and sampling could happen continuously throughout the run of the program to optimally execute long-running programs, tuning throughout the run of the program.

The Model might also be brought to address nodes of multiple architectures in the same cluster. For example, if we define our "cluster" to be an SGI-Cray T3E, an SGI-Cray T90, and IBM SP, connected by a high-bandwidth, low-latency network, we will have a *phenomenally heterogeneous cluster* or *PHC* (as opposed to a *slightly heterogenous cluster* or *SHC*). A problem that is able to make use of all of these machines and their unique characteristics would be rare, but it is entirely possible that a program might have some loops that are easily vectorizable and should best be directed to the T90 and parts of the program that simply should be farmed out to as many processors as possible on the T3E and SP [5]. A

possible step for the ClusterDiscovery stage would be to recognize this and direct the ClusterOptimizer to divide not necessarily just the data or the computation as a whole but to separate the problem into specific tasks that could be assigned to each unique hardware architecture according to the machines' specialties.

I would like the readers of this thesis to bring away with them, the following points:

1. Sputnik is an API which demonstrates that heterogeneous clusters of multiprocessors need not be difficult to program. As clusters of multiprocessors appear to be the near-term future for supercomputing, ways are needed to address the evolution of these machines. Sputnik is one of these ways.

2. Sputnik is an API which demonstrates that programs need not waste any available processing power on a heterogeneous cluster of multiprocessors. By adjusting the number of threads per multiprocessors and repartitioning the problem so that each multiprocessors node is time-balanced, the program can run as if the entire cluster was homogeneous.

3. Sputnik can be extended in the future, as I intend to do myself in many ways in future research with the San Diego Supercomputer Center, including, but not limited to:

   (a) Running on a variety of vendor platforms.

   (b) Performing dynamic optimizations.

   (c) Running on clusters of varying architectures (including vector machines and MPPs), not just varying speed, memory, network, or cache size of multiprocessors.

   (d) Performing varying types of optimizations — more than just repartitioning or adjusting the number of threads.

   (e) Working with many other different types applications.

Contact Inforamtion

The author can be contacted at the email address:

peisert@sdsc.edu

This thesis can be downloaded in full, in PDF format, at:

http://www.sdsc.edu/~peisert/

Appendix A

User's Guide

A.A Introduction

Sputnik runs in two stages. In the first stage, the routines gather information about how the program actually runs on each multiprocessor node. Once it gets timings for each separate node, it calculates the fraction of the workload that each multiprocessor node should run. This ratio looks like this: $time_n/total\_time = work_n/total\_work$ in a cluster with N multiprocessor nodes. Additionally, the first stage determines the optimal number of OpenMP threads to run per node. This particular optimization of the ideal number of threads per node to run is only one possible optimization that we could be doing. Other possible optimizations might include tiling for cache and making predictions about dynamic optimizations that the program might need after the original partitioning.

In the second stage, the routines partition the problem based on the calculated fractions, and finally make a final run of the program at the peak speed possible based on the chosen partition.

Sputnik expands upon KeLP1 by adding to the API with two new functions and one new technology, OpenMP. Further, Sputnik modifies the existing KeLP distribution by adding new partitioning algorithms to the *distribute* and *distributeBlock1* functions in the DOCK DecompositionX class. The goal of the

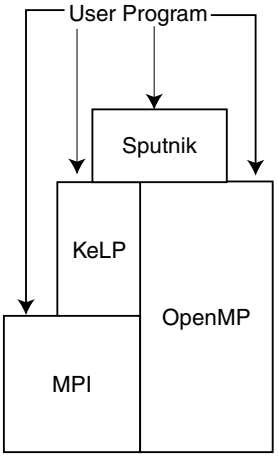


Figure A.1: Hierarchy of software layers for Sputnik

API, designed to be consistent with the original KeLP1 goals of making scientific program easier while making the overall performance greater has resulted in an API that requires only a few minor modifications to an existing KeLP1 program to work in Sputnik. A good progression for writing a Sputnik program would be to first write the serial program, then modify it to be a KeLP1 program, then add OpenMP directives, and then finally modify the KeLP1 program to be a Sputnik program using the Sputnik API calls.

A.B Major Changes in Usage from KeLP1

The two new routines are as follows (which are described in detail later in this chapter):

```
void SputnikGo(int argc, char **argv)
double SputnikMain(int argc, char **argv, double * SputnikTimes)
```

Sputnik accepts more arguments than those that may just be passed into the application or to MPI. A typical MPI program, called *mpiprog* in this example, is started like this on four nodes: `mpirun -np 4 mpiprog`.

Sputnik takes four additional arguments:

1. testSMPS: If equal to the integer "1", this tells the program that it should pay attention to the "maxthreads" argument, ignore the "numthreads" and "hetero" arguments and that it should test the speed of each individual multiprocessor node before making a final run with the optimal number of threads per node. If equal to the integer "0", the "hetero" argument is checked. If "hetero" is equal to "1," then the ".stats" file is read for its timing information and threads are allocated to the nodes based on the timings from ".stats" and the value of "maxthreads." If "hetero" is equal to 0, then the ".stats" file and "maxthreads" are ignored and "numthreads" threads per node are used rather than reading the timings from the ".stats" file. All other conditions will produce an error.

2. hetero: Whether or not to run heterogeneously, based on the ".stats" file, if the multiprocessor nodes are not tested individually.

3. maxthreads: If given, is the maximum number of threads to be used on each node. If not given, a default value is used.

4. numthreads: The number of threads per node to allocate if testSMPS is 0 (false).

A.C Examples of Command-Line Options

An example of each possible running mode with the Sputnik command-line arguments follows. Each example runs an MPI program *mpiprog* with a certain number of threads on two multiprocessor nodes.

1. `mpirun -np 2 mpiprog -testSMPS 1 -maxthreads 10`

    - With a maximum number of 10 threads on both nodes, Sputnik will test each node with the kernel of the program to determine the optimal number of threads for each node. When it has found the optimal times,

it writes them to a file called ".stats" and makes a final run of the program using the optimal number of threads per node and the optimal decomposition.

- The `-hetero` argument is ignored.

2. `mpirun -np 2 mpiprog -testSMPS 1 -maxthr0 10 -maxthr1 20`

   - With a maximum number of 10 threads on node 0 and 20 threads on node 1, Sputnik will test each node with the kernel of the program to determine the optimal number of threads for each node. When it has found the optimal times, it writes them to a file called ".stats" and makes a final run of the program using the optimal number of threads per node and the optimal decomposition.
   - The `-hetero` argument is ignored.

3. `mpirun -np 2 mpiprog -testSMPS 0 -hetero 0 -numthreads 10`

   - Sputnik runs on 2 nodes with exactly 10 threads per node with homogeneous decomposition.

4. `mpirun -np 2 mpiprog -testSMPS 0 -hetero 0`
   `-numthr0 10 -numthr1 20`

   - Sputnik runs on 2 nodes with exactly 10 threads on node 0 and 20 threads on node 1 with homogeneous decomposition. Use of this set of options is comparable to running with KeLP2.

5. `mpirun -np 2 mpiprog -testSMPS 1 -hetero 0`
   `-numthr0 10 -numthr1 20`

   - Sputnik runs on 2 nodes with exactly 10 threads on node 0 and 20 threads on node 1, but with heterogeneous decomposition based on a single test run made before the final run.

6. `mpirun -np 2 mpiprog -testSMPS 0 -hetero 1 -numthreads 10`

   - Sputnik runs on 2 nodes with exactly 10 threads per node with heterogeneous decomposition based on the times stored in the ".stats" file.

7. `mpirun -np 2 mpiprog -testSMPS 0 -hetero 1 -numthr0 10 -numthr1 20`

   - Sputnik runs on 2 nodes with exactly 10 threads on node 0 and 20 threads on node 1 with heterogeneous decomposition based on the times stored in the ".stats" file.

A.D Example of Intra-Node Parallelism

Additionally, OpenMP directives must be put around the loops that the programmer wishes to parallelize. The directives function either in C/C++ or Fortran kernels and can be as simple as those used in this code from a KeLP version of Red-Black 3D:

```
!$OMP PARALLEL PRIVATE(jj,ii,k,j,i,jk)
 do jj = ul1+1, uh1-1, sj
   do  ii = ul0+1, uh0-1, si
!$OMP DO SCHEDULE(STATIC)
     do k = ul2+1, uh2-1
       do  j = jj, min(jj+sj-1,uh1-1)
         jk = mod(j+k,2)
         do  i = ii+jk, min(ii+jk+si-1,uh0-1), 2
            u(i,j,k) = c *
     2      ((u(i-1,j,k) + u(i+1,j,k)) + (u(i,j-1,k) +
     3      u(i,j+1,k)) + (u(i,j,k+1) + u(i,j,k-1) -
     4      c2*rhs(i,j,k)))
```

```
    end do
       end do
     end do
!$OMP END DO
    end do
  end do

!$OMP END PARALLEL
```

The code, above, scales well and with the OpenMP directives in place can show parallelism well, depending on the size of the problem.

A.E Sputnik Usage

Sputnik works in this manner:

1. Initialize MPI and KeLP
2. Set the number of threads on each multiprocessor node and run the kernel of the program repeatedly on each individual multiprocessor node, varying the number of threads, until the optimal number of threads per node is reached. The kernel runs in a kind of "shell" that Sputnik creates using SputnikMain() (described below).
3. Run the kernel with the optimal number of thread per node.

In order to use the Sputnik library, there are three principle changes that need to be made to the KeLP code in addition to adding OpenMP directives. First, a new function needs to be defined by the programmer, `SputnikMain()`. Second, all calls to the `distribute` function from the user code need to have an additional argument added. Finally, the main Sputnik function, `SputnikGo`, needs to be called by the programmer from within `main`.

A.E.1 SputnikMain()

`SputnikMain()` is the code that is called over and over again while trying to find the optimal number of threads per multiprocessor node. It is not just the kernel of the program, but is also everything that the program needs to call before running the kernel, such as the initialization of values in arrays. `SputnikMain` returns a double. The value of that double should be the time it takes for the kernel to run. For example:

```
double SputnikMain(int argc,char ** argv, double * SputnikTimes) {
    double start, finish;

    ...
    <declaraitions, initializations>
    ...
    start = MPI_Wtime();  // start timing
    kernel();             // call the kernel function
    finish = MPI_Wtime(); // finish timing
    ...
    return finish-start;
}
```

Essentially everything that was in `main()` can now be in `SputnikMain()` with the addition of timing calls. A bare-bones `main()` might now look like this:

```
int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv); // Initialize MPI
   InitKeLP(argc,argv);    // Initialize KeLP

   // Call Sputnik's main routine, which in turn will
```

```
   // then call SputnikMain().
   SputnikGo(argc,argv);

   MPI_Finalize();         // Close off MPI
   return (0);

}
```

The call that sets the number of threads per node to use is actually set in SputnikGo() and so does not need to be used by the programmer. The number of threads actually being used in a loop should be tested by using the OMP_GET_MAX_THREADS() call. In this way, the programmer can determine whether OpenMP is doing a good job of parallelizing the kernel that the programmer would like to speed up. A number of factors can affect OpenMP's parallelization, including striding of loops and data dependencies.

A.F Sputnik Implementation

The overall process of what Sputnik does has been discussed previously in this thesis. I will discuss the specifics here. Sputnik has many command-line options, but in this following section, I will discuss just the part where we test the strength of the multiprocessor nodes (`-testSMPS = 1`), which is the most important and unique part of Sputnik's functionality.

After `SputnikGo()` is called (probably from within `main()`), the program follows something similar to this outline in pseudo-code:

```
// Until we have hit the user-defined limit of the maximum number
// of threads or until the time we get by increasing the number of
// threads is higher (worse performance) than the lower number of
// threads, keep increasing the number of threads and running the
```

```
// kernel of the program, as contained in SputnikMain(), without
// communication.  This way, we can get the individual timings for
// each multiprocessor node.
while(i < MAX_THREADS
        && time[last iteration] < time[second-to-last iteration])
{
    omp_set_num_threads(i);

    // By passing in NULL for the times, we are telling the
    // routine inside the DecompositionX class not to do any
    // special modifications.
    time[i] = SputnikMain(int argc, char **argv, NULL);
    i = i * 2;
}

i = iteration before the best we found in the previous loop;

// During the first loop, we move quickly to find the optimal
// solution by doubling i each time.  This time, we only
// increment by 1, starting with the best estimate from the
// first while loop.
while (time[last iteration] < time[second-to-last iteration])
{
    omp_set_num_threads(i);
    time[i] = SputnikMain(int argc, char **argv, NULL);
    i = i + 1;
}

// Set the optimal number of threads.  Each node may have a
```

```
// different optimal value.
omp_set_num_threads(optimal number);

// This time, pass in the best times and let the DecompositionX
// routines do partitioning based on the best times for each node.
// This way, not only does each node run with an optimal number
// of threads per node (it may not be the maximum available), but
// also with an optimal division of work.
time[i] = SputnikMain(int argc, char **argv, bestTimes);
```

Appendix B

Source Code to Sputnik

B.A DecompositionX.h.m4

```
/*****************************************************************
*   DecompositionX.h.m4                                          *
*                                                                *
*   Author: Stephen Fink                                         *
*   Modified for Sputnik: Sean Peisert                           *
*                                                                *
*   Class DecompositionX represents a distributed index space    *
*   with a regular block decomposition                           *
*****************************************************************/

#include "ArrayX.h"
#include "ProcessorsX.h"
#include "GhostPlanX.h"
#include "menagerie.h"

#define BLOCK1 1
#define BLOCK2 2

/*****************************************************************
*   Class DecompositionX is a first-class dynamic template  *
*****************************************************************/
class DecompositionX: public GhostPlanX {
   RegionX _domain;    /* Global region of DecompositionX      */
   PointX _distType;   /* Distribution directive in each
                          dimension */
   ArrayX<int> _Map;   /* maps virtual proc array to 1-d
```

```
                          floorplan */
public:
   /*********************************/
   /* constructors and destructors */
   /*********************************/
   DecompositionX() {}
   DecompositionX(ArrayIndexArguments):
    _domain(PointX(1),PointX(ArrayIndices)) {}
   DecompositionX(const RegionX& R): _domain(R) {}
   DecompositionX(const DecompositionX& D);

   DecompositionX& operator = (const DecompositionX& D);

   /*********************************/
   /* simple access functions      */
   /*********************************/
   const PointX& distributionRules() const {return _distType;}
   const RegionX& domain() const {return _domain;}
   int domainEmpty() const {return _domain.empty();}
   int domainLower(const int dim) const
                               {return _domain.lower(dim);}
   int domainUpper(const int dim) const
                               {return _domain.upper(dim);}
   int domainExtents(const int dim) const
                               {return _domain.extents(dim);}
   int domainSize() const {return _domain.size();}

   // query functions about the virtual processor array
   int pLower(const int dim) const {return _Map.lower(dim); }
   int pUpper(const int dim) const {return _Map.upper(dim); }
   int pExtents(const int dim) const {return _Map.extents(dim); }
   int pMap(const PointX& P) const { return _Map(P); }

   // query functions about the global index domain
   int pIndex(const int dim, const int gIndex) const;
   int pOwner(const PointX& P) const;
   RegionX pRegion(const RegionX& R) const;

   const XObjectX& operator () (const int i) const
   { return FloorPlanX::operator() (i);         }
   const XObjectX& operator () (const PointX& P) const
   { return (*this)(pMap(P));               }
```

```
   void setowner(const int i, const int proc)
   { FloorPlanX::setowner(i,proc);          }
   void setowner(const PointX& P, const int proc)
   { FloorPlanX::setowner(pMap(P),proc);        }

   void setDomain(const RegionX& D)
   {    _domain = D;                        }

   /*******************************/
   /* distribution methods         */
   /*******************************/
   void distribute(const PointX& D,const ProcessorsX& P,
                   double * Times);
   void distribute(ArrayIndexArguments, const ProcessorsX& P,
                   double * Times)
     { distribute(PointX(ArrayIndices),P,Times); }
   void distribute(const PointX& D, double * Times)
     { distribute(D,ProcessorsX(),Times); }
   void distribute(ArrayIndexArguments, double * Times)
     { distribute(PointX(ArrayIndices),ProcessorsX(),Times); }

private:
   /* simple access functions */
   int procExtents(int dim) const
    {return _Map.extents(dim);}
   int procLower(int dim) const
    {return _Map.lower(dim);}

   /* distribution functions */
   void distributeBlock1(int dim, double * Times);
   void distributeBlock2(int dim);
};

#endif
```

B.B DecompositionX.C.m4

B.B.1 distribute

```
/*****************************************************************
* void DecompositionX::distribute(const PointX& D,              *
*               const ProcessorsX& P)                           *
*                                                               *
```

```
* Distribute a decomposition across the logical processor array.*
*****************************************************************/
void DecompositionX::distribute(const PointX& D,
                                const ProcessorsX& P,
                                double * Times)
{
   /* initialize the _Map array */
   _Map.resize(P.region());
   resize(_Map.size());

   int i = 0;
   for_point_X(p,_Map)
      _Map(p) = i;
      i++;
   end_for

   if (domainEmpty()) return;

   for (int dim=0;dim<NDIM;dim++) {
     switch(D(dim)) {
       case BLOCK1:
      distributeBlock1(dim, Times);
      break;
       case BLOCK2:
      distributeBlock2(dim);
      break;
       default:
      break;
     }
   }

   /* do processor assignments */
   for_point_X(p,_Map)
     setowner(_Map(p),P(p));
   end_for
}
```

B.B.2 distributeBlock1

```
/*****************************************************************
*  void DecompositionX::distributeBlock1(int dim)                *
*                                                                *
```

```
*  In a BLOCK1 distribution, each procesor gets exactly         *
*  ceiling(N/P) elements. If N doesn't divide P, this will      *
*  result in a load imbalance.                                  *
****************************************************************/

void DecompositionX::distributeBlock1(int dim, double * Times)
{
   int N = domainExtents(dim);
   int P = pExtents(dim);
   int PLower = pLower(dim);
   int dimOffset, low;
   int ceiling;

   int i;
   double tTotal=0.0;
   double * invTimes = new double[P];
   int * ceilings = new int[P];
   int * aHigh = new int[P];

   if (Times != NULL && P > 1) {
     for (i = 0; i < P; i++) {

        /* Get the inverses of the total times */
        tTotal += 1.0/Times[i];
        invTimes[i] = 1.0/Times[i];
     }
     for (i = 0; i < P; i++) {
        /* Get the ratios and even it out, if necessary */
        ceilings[i] = floor((invTimes[i]/tTotal)*N);
        if (N%P == 0)
            ceilings[i] += 1;
     }
   }
   else {
     ceiling = (N%P)? (N/P)+1 : N/P;
   }


   _distType(dim) = BLOCK1;

   i = 0;
   for_point_X(p,_Map)
     dimOffset = p(dim) - PLower;
```

```
     if (Times != NULL && P > 1) {
        ceiling = ceilings[i];
        if (i > 0) {
            low =  aHigh[i-1] + 1;
        }
        else {
            low =  domainLower(dim) + ceiling * dimOffset;
        }
        aHigh[i] = low + ceiling - 1;
        i++;
     }
     else {
        low =  domainLower(dim) + ceiling * dimOffset;
     }
     setlower(_Map(p),dim,low);
     setupper(_Map(p),dim,MIN(low + ceiling - 1,
              domainUpper(dim)));
   end_for
}
```

B.C Sputnik.h

```
void SputnikGo(int argc, char **argv);
double SputnikMain(int argc, char **argv, int testSMPS,
                   double * SputnikTimes);
```

B.D Sputnik.C

```
#include <iostream.h>
#include "Sputnik.h"
#include <omp.h>
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define def_numThreads 10;
#define def_maxThreads 0;
#define def_testSMPS 0;
#define def_hetero 0;
```

```
void SputnikGo(int argc, char **argv) {
       int numThreads;
       int hetero;
       int maxThreads;
       int testSMPS;
       int maxThr[2], numThr[2];

       int i, j, k;
       int myid, nodes;
       char procName[80];
       MPI_Comm_rank(MPI_COMM_WORLD,&myid);
       MPI_Comm_size(MPI_COMM_WORLD,&nodes);

       if (myid == 0) {
        testSMPS = def_testSMPS;
        testSMPS = def_hetero;
        numThreads = def_numThreads;
        maxThreads = def_maxThreads;
        maxThr[0] = 0;
        maxThr[1] = 0;
        numThr[0] = 0;
        numThr[1] = 0;

        for (int arg = 1; arg < argc; arg++) {
         if (!strcmp("-testSMPS",argv[arg]))
                testSMPS = atoi(argv[++arg]);
         else if (!strcmp("-hetero",argv[arg]))
                hetero = atoi(argv[++arg]);
         else if (!strcmp("-numthreads",argv[arg]))
                numThreads = atoi(argv[++arg]);
         else if (!strcmp("-maxthreads",argv[arg]))
                maxThreads = atoi(argv[++arg]);
         else if (!strcmp("-maxthr0",argv[arg]))
                maxThr[0] = atoi(argv[++arg]);
         else if (!strcmp("-maxthr1",argv[arg]))
                maxThr[1] = atoi(argv[++arg]);
         else if (!strcmp("-numthr0",argv[arg]))
                numThr[0] = atoi(argv[++arg]);
         else if (!strcmp("-numthr1",argv[arg]))
                numThr[1] = atoi(argv[++arg]);
        }
         cout << "INPUTS: " << endl;
        cout << "\ttestSMPS: " << testSMPS << endl;
```

```
 cout << "\thetero: " << hetero << endl;
 cout << "\tnumthreads: " << numThreads << endl;
 cout << "\tmaxthreads: " << maxThreads << endl;
 cout << "\tmaxthreads for SMP 0: " << maxThr[0] << endl;
 cout << "\tmaxthreads for SMP 1: " << maxThr[1] << endl;
 cout << "\tnumthreads for SMP 0: " << numThr[0] << endl;
 cout << "\tnumthreads for SMP 1: " << numThr[1] << endl;

 for (i = 0; i < nodes; i++) {
        k = MPI_Get_processor_name(procName,&j);
       if(!k)
               cout << "\tname of SMP " << i << ": "
                    << procName << endl;
 }

 if (testSMPS == 1 && maxThreads > 0) {
        cout << "TEST MODE ON\n";
       if (maxThr[0] != 0 && maxThr[1] != 0) {
              cout << "MAX THREADS SET INDIVIDUALLY\n";
       }
       else
              cout << "MAX THREADS SET AS A GROUP\n";
 }
 else if (testSMPS == 1
         && (numThreads > 0 || numThr[0] > 0)
         && maxThreads == 0) {
        cout << "TEST MODE ON\n";
       if (numThr[0] != 0 && numThr[1] != 0)
        cout << "SPECIFIC NUM THREADS SET INDIVIDUALLY\n";
       else
        cout << "SPECIFIC NUM THREADS SET AS A GROUP\n";
 }
 else {
       cout << "FIXED NUMBER OF THREADS\n";
       if (numThr[0] != 0 && numThr[1] != 0)
        cout << "SPECFIC NUM THREADS SET INDIVIDUALLY\n";
       else
        cout << "SPECIFIC NUM THREADS SET AS A GROUP\n";
 }
}
k = MPI_Get_processor_name(procName,&j);
if(!k)
 cout << "\tname of SMP " << myid << ": "
```

```
      << procName << endl;

int bestRun;
double bestTime;
double final;
double * bestTimes = new double[nodes];
double * staticTimes = new double[nodes];
int * bestRuns = new int[nodes];
FILE * stats;

MPI_Barrier(MPI_COMM_WORLD);
// MPI problem: only root process gets argv correctly
// must broadcast ring size to other processes
MPI_Bcast(&testSMPS,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&hetero,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&numThreads,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&maxThreads,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&maxThr,2,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&numThr,2,MPI_INT,0,MPI_COMM_WORLD);

if (maxThr[0] != 0 && maxThr[1] != 0)
       maxThreads = maxThr[myid];

int maxMax = maxThreads;

// Get the biggest number of threads on all nodes
for (i = 0; i < nodes; i++) {
       if (maxMax < maxThr[i])
              maxMax = maxThr[i];
}

if (testSMPS == 1 && maxThreads > 0) {
       double * timings = new double[maxMax+1];
       // Initialize Values to Zero.  We can't use zero
       // threads, so we go all the way up to maxThreads
       for (i = 0; i <= maxMax; i++) {
              timings[i] = 0.0;
       }

       int gotit = 0;
       int * a_gotit = new int[nodes];
       int * a_min = new int[nodes];
       int * a_max = new int[nodes];
```

```
int * a_iter = new int[nodes];
int minMin=1;

for (i = 0; i < nodes; i++) {
       bestTimes[i] = 0.0;
       bestRuns[i] = 0;
       a_gotit[i] = 0;
       a_min[i] = 0;
       a_max[i] = 0;
}

i = 1;

// Find the Max and Min times that the best
// is in-between
while (i <= maxMax) {
       cout << "PE "<< myid << ": Running with "
            << i << " threads.\n";
       MPI_Barrier(MPI_COMM_WORLD);
       a_iter[myid] = i;
       for (j = 0; j < nodes; j++) {
              MPI_Bcast((void *) &a_iter[j],1,
                        MPI_INT,j,MPI_COMM_WORLD);
              assert(a_iter[j] == i);
       }
       omp_set_num_threads(i);

       // Initial run
       timings[i] = SputnikMain(argc, argv,
                                testSMPS, NULL);

       // Store the best time for the first run
       // or a lower time
       if ((timings[i] <= bestTime || i == 1)
                     && gotit == 0
                     && i <= maxThr[myid]) {
              bestRun = i;
              bestTime = timings[i];
        }

       // If the time goes up
       if ((timings[i] > bestTime) && gotit == 0){
              a_min[myid] = i/4;
```

```
                a_max[myid] = i;
                gotit = 1;
                a_gotit[myid] = 1;
        }

        //keep increasing number of threads
        if (i*2 > maxMax && i != maxMax) {
                if (gotit == 0) {
                        // 2nd-highest power of 2
                        // before maxMax
                        a_min[myid] = i/2;

                        //maxMax
                        a_max[myid] = maxMax;
                }
                i = maxMax;
        }
        else {
                i*=2;
        }

        // Check to see if everyone's time went up
        int allgotit = 0;
        for (j = 0; j < nodes; j++) {
                MPI_Bcast((void *) &a_gotit[j],1,
                          MPI_INT,j,MPI_COMM_WORLD);
                allgotit +=a_gotit[j];
        }

        if (a_max[myid] == maxThr[myid]) {
                a_min[myid] = maxThr[myid]/2;
                gotit = 1;
                a_gotit[myid] = 1;
        }
}

minMin = a_min[myid];
maxMax = a_max[myid];
for (j = 0; j < nodes; j++) {
        MPI_Bcast((void *) &a_min[j],1,MPI_INT,
                  j,MPI_COMM_WORLD);
        if (a_min[j] < minMin)
                minMin = a_min[j];
```

```
}
for (j = 0; j < nodes; j++) {
       MPI_Bcast((void *) &a_max[j],1,MPI_INT,
                 j,MPI_COMM_WORLD);
       if (a_max[j] > maxMax)
              maxMax = a_max[j];
}

if (minMin <= 0)
       minMin = 1;

i = minMin;

// Step through from the min to the max
// to find the best
while (i <= maxMax) {

       if (timings[i] != 0.0) {
              i++;
              continue;
       }
       cout << "PE "<< myid << ": Running with "
            << i << " threads.\n";

       omp_set_num_threads(i);

       timings[i] = SputnikMain(argc, argv,
                                testSMPS, NULL);

       if ((timings[i] <= bestTime)
           && i <= maxThr[myid]) {
              bestTime = timings[i];
              bestRun = i;
       }
       i++;
}

MPI_Barrier(MPI_COMM_WORLD);

bestTimes[myid] = bestTime;
bestRuns[myid] = bestRun;

for (i = 0; i < nodes; i++) {
```

```
                MPI_Bcast((void *) &bestTimes[i],1,
                          MPI_DOUBLE,i,MPI_COMM_WORLD);
                MPI_Bcast((void *) &bestRuns[i],1,
                          MPI_INT,i,MPI_COMM_WORLD);
        }

        if (myid == 0) {

                stats = fopen("StatPut","w");
                for (i = 0; i < nodes; i++) {
                        fprintf(stats,"%f\n",bestTimes[i]);
                        cout << "bestRuns (node " << i
                             << "): " << bestRuns[i] << endl;
                        cout << "bestTimes (node " << i
                             << "): " << bestTimes[i] << endl;
                }

                 for (i = 0; i < maxThreads+1; i++)
                        fprintf(stats,
                                "Time for %d threads: %f\n",
                                i,timings[i]);
                fclose(stats);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        testSMPS = 0;
        omp_set_num_threads(bestRuns[myid]);
        final = SputnikMain(argc, argv,
                            testSMPS, bestTimes);
}
else if (testSMPS == 1
         && (numThreads > 0 || numThr[0] > 0)
         && maxThreads == 0) {

        if (numThr[0] != 0) {
                omp_set_num_threads(numThr[myid]);
                cout << "SETTING ID THREADS " << myid << ","
                     << numThr[myid] << endl;
        }
        else {
                omp_set_num_threads(numThreads);
                cout << "SETTING ID THREADS " << myid << ","
                     << numThreads << endl;
        }
```

```
        MPI_Barrier(MPI_COMM_WORLD);
        bestTimes[myid] = SputnikMain(argc, argv,
                                      testSMPS, NULL);
        MPI_Barrier(MPI_COMM_WORLD);

        for (i = 0; i < nodes; i++) {
               MPI_Bcast((void *) &bestTimes[i],1,
                         MPI_DOUBLE,i,MPI_COMM_WORLD);
        }

        if (myid == 0) {
               for (i = 0; i < nodes; i++) {
                      cout << "bestTimes (node " << i
                           << "): " << bestTimes[i] << endl;
               }
        }
        MPI_Barrier(MPI_COMM_WORLD);

        testSMPS = 0;
        final = SputnikMain(argc, argv, testSMPS, bestTimes);
}
else { // testSMPS == 0
        if (numThr[0] != 0)
               omp_set_num_threads(numThr[myid]);
        else
               omp_set_num_threads(numThreads);

        if(hetero) {
               if (myid == 0)
                      cout << "TIMES TO PARTITION FOR:\n";
               stats = fopen("StatPut","r");
               for (j = 0; j < nodes; j++) {
                      fscanf(stats, "%lf\n",
                             &(staticTimes[j]));
                      if (myid == 0)
                             cout << staticTimes[j]
                                  << " seconds\n";
               }
               fclose(stats);

               final = SputnikMain(argc, argv,
                                   testSMPS, staticTimes);
```

```
                }
                else {
                        // Run "unaltered"
                        final = SputnikMain(argc, argv,
                                                testSMPS, NULL);
                }
        }
        MPI_Barrier(MPI_COMM_WORLD);
        cout << "Final (" << myid << "): " << final << endl;
        cout << "Done." << endl;
        cout.flush();
        MPI_Barrier(MPI_COMM_WORLD);
}
```

Appendix C

Source code to redblack3D with Sputnik

C.A rb.F

```
c*************************************************************
c       subroutine rb7rrelax(u,ul0,ul1,ul2,uh0,uh1,uh2)
c       integer ul0, ul1,uh0, uh1, ul2, uh2
c       double precision u(ul0:uh0,ul1:uh1,ul2:uh2)
c
c       peform 7-point red/black relaxation for Poissons's
c       equation with h=1.0
c
c        Originally written by Stephen J. Fink,
c        Modified by Scott B. Baden for 3D RB
c              Converted to 3D
c              Blocked for Cache
c              RB ordering
c*************************************************************

c       Smooth the Red Points
       subroutine rb7rrelax(u,ul0,ul1,ul2,uh0,uh1,uh2,si,sj,rhs)
       integer ul0, ul1,uh0, uh1, ul2, uh2, si, sj
       double precision u(ul0:uh0,ul1:uh1,ul2:uh2)
       double precision rhs(ul0:uh0,ul1:uh1,ul2:uh2)
       double precision c,h,c2

       integer i, j, k, ii, jj, jk

       c= (1.0d0/6.0d0)
       h=1.0d0
```

```
       c2=h*h

!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(jj,ii,k,j,i,jk)
!$OMP DO SCHEDULE(STATIC)
       do jj = ul1+1, uh1-1, sj
          do  ii = ul0+1, uh0-1, si
          do k = ul2+1, uh2-1
                do  j = jj, min(jj+sj-1,uh1-1)
                   jk = mod(j+k,2)
                    do  i = ii+jk, min(ii+jk+si-1,uh0-1), 2
                      u(i,j,k) = c *
     2                      ((u(i-1,j,k) + u(i+1,j,k)) +
     3                         (u(i,j-1,k) + u(i,j+1,k)) +
     4                         (u(i,j,k+1) + u(i,j,k-1) -
     5                         c2*rhs(i,j,k)))
                   end do
                end do
             end do
          end do
       end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL

       return
       end

c       Smooth the black points
       subroutine rb7brelax(u,ul0,ul1,ul2,uh0,uh1,uh2,si,sj,rhs)
       integer ul0, ul1,uh0, uh1, ul2, uh2, si, sj
       double precision u(ul0:uh0,ul1:uh1,ul2:uh2)
       double precision rhs(ul0:uh0,ul1:uh1,ul2:uh2)
       double precision c,c2,h

       integer i, j, k, ii, jj, jk
       c= (1.0d0/6.0d0)
       h=1.0d0
       c2=h*h

!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(jj,ii,k,j,i,jk)
!$OMP DO SCHEDULE(STATIC)
       do jj = ul1+1, uh1-1, sj
          do  ii = ul0+1, uh0-1, si
          do k = ul2+1, uh2-1
```

```
               do  j = jj, min(jj+sj-1,uh1-1)
                  jk = 1 - mod(j+k,2)
                   do  i = ii+jk, min(ii+jk+si-1,uh0-1), 2
                     u(i,j,k) = c *
     2                    ((u(i-1,j,k) + u(i+1,j,k)) +
     3                    (u(i,j-1,k) + u(i,j+1,k)) +
     4                    (u(i,j,k+1) + u(i,j,k-1) -
     5                    c2*rhs(i,j,k)))
                  end do
               end do
            end do
         end do
       end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL

       return
       end
```

C.B rb3D.C

```
/*********************************************************************
* rb3D.C                                                             *
*                                                                    *
* program that solves Poisson's equation on a unit cube              *
* using Red/Black ordering                                           *
* We should never solve Poisson's equation this way;                 *
* this kernel is intended to be used in multigrid                    *
*                                                                    *
*                                                                    *
*  This version uses a modified custom MotionPlan to send            *
*  contiguous messages where possible.                               *
*  It also optimizes communication still further using the           *
*  the Manhattan class to avoid communicating corner and             *
*  edge ghost cells; the solver uses a 7 point stencil so            *
*  there is no need to send these extra points.                      *
*  If you use this code as a starting point for another              *
*  applcation, and you need the corner or edge points, do not        *
*  use the Manhattan class: use an IrregularGrid3 instead.           *
*  The code may be easily modified to this end                       *
*  Replace Manhattan by IrregularGrid3 and be sure to uncomment      *
*  the code that sets up the Mover and MotionPlan objects            *
```

```
*                                                                     *
*    Uncomment the following Mover member function calls:             *
*       execute()                                                     *
*                                                                     *
*    Comment out the following Manhattan member function calls        *
*       fillGhost()                                                   *
*       Optimze()                                                     *
*                                                                     *
* Original 2D code was written by Stephen J. Fink                     *
* Extensively modified for benchmarking by Scott B. Baden             *
* Deptartment of Computer Science and Engineering,                    *
* University of California, San Diego                                 *
*                                                                     *
**********************************************************************/

#include "Sputnik.h"
#include "j3D.h"
#include "XArray3.h"
#include "Grid3.h"
#include "Mover3.h"
#include "timer.h"
#include "manhattan.h"
#include <omp.h>

//extern int testSMPS;

void cmdLine(int argc, char **argv,
             int&L, int& M, int& N, double& eps, int& niter,
            int& chk_freq, int& reps, int& si, int& sj, int&gi,
             int&gj);

void ReportTimings(double times[],double timesLoc [],
             int reps, int chk_freq, int niter, int N,
             int si, int sj, int gi, int gj);

void InitGrid(IrregularGrid3& grid)
{
   grid.fill(1.0);
   grid.assignGhost(0.0);
}
```

```
void ComputeLocal(IrregularGrid3& grid,
                int si, int sj, const int color,
                IrregularGrid3& rhs);

/*************************************************************
* main()                                                     *
*                                                            *
* main() takes one argument: N: the number of points         *
* along each axis                                            *
*************************************************************/
int main(int argc,char **argv)
{
   MPI_Init(&argc, &argv);

   SputnikGo(argc,argv);

   MPI_Finalize();
   return (0);

}
double SputnikMain(int argc,char ** argv,
                   int testSMPS, double * SputnikTimes) {
       double start, finish, middle;
    middle = 0.0;

   try {

   InitKeLP(argc,argv);
   KelpConfig(CONTIG_MSG_IN_PLACE,TRUE);

   int L,M,N, chk_freq, niter, reps, si, sj, gi, gj;
   double eps;
   cmdLine(argc, argv, L,M, N, eps, niter,chk_freq, reps,
           si, sj, gi, gj);

   Region3 domain(1,1,1,N,N,N);

   /* Print header information*/
   OUTPUT("rb3D run on P = " << mpNodes()
          << " nodes with N = " << N
   << endl);
   OUTPUT("Processors geometry: "<< gi << " x " << gj << " x "
          << mpNodes()/(gi*gj) << "   Blocking factors: " << si
```

```
        << " x " << sj << endl);
    OUTPUT("# Iter = " << niter << "  # Reps: "  << reps << endl);
    if (chk_freq <= niter)
     OUTPUT("     Convg check every " << chk_freq
            << " Iterations");

    /* Allocate space for the local part of the problem */
    /* Use the dock library to help with the partitioning */

    const Region3 STRIP_V(1,1,1,gi,gj,mpNodes()/(gi*gj));
    Processors3 P(STRIP_V);
    OUTPUT(P << endl);

    Decomposition3 T(domain);
    T.distribute(BLOCK1,BLOCK1,BLOCK1,P, SputnikTimes);
    OUTPUT(T << endl);

    T.addGhost(1);
    Manhattan grid(T);
    IrregularGrid3 rhs(T);

// Initialize the local grid
    InitGrid(grid);
    rhs.fill(0.0);

    IrregularGrid3 *U = &grid;

    double stop = 1.0;
    double* times = new double[reps];
    double* timesLoc = new double[reps];

    const int RED = 0 , BLK = 1;

    //
    // Do the computation
    // We actually do it twice: once with communication,
    // and once without
    //

    if (mpNodes() > 1 && !testSMPS){
        for (int k = 0 ; k < reps; k++){
           STATS_RESET();
           STATS_START(STATS_ITERATION);
```

```
            if (k != 0) {
                      start = MPI_Wtime();
            }

          for (int i= 1; i<=niter; i++) {

            // Exchange boundary data with neighboring
            // processors
            U->fillGhost();

            // Perform the local smoothing operation on the
            // RED Points
            ComputeLocal(*U,si,sj,RED,rhs);

            // Exchange boundary data with neighboring
            // processors
            U->fillGhost();

            // Perform the local smoothing operation on the
            //    RED Points
            // Perform the local smoothing operation on the
            //    BLK Points
            ComputeLocal(*U,si,sj,BLK,rhs);

          }
            if (k != 0) {
                      finish = MPI_Wtime();
                     middle = finish - start + middle;
            }
           STATS_STOP(STATS_ITERATION);
          times[k] = STATS_TIME(STATS_ITERATION);
      }
            if(!testSMPS)
                   cout << "SPUTNIK TIME WITH COMMUNICATION: "
                        << middle << endl;
}

      middle = 0.0;
for (int k = 0 ; k < reps; k++){
      STATS_RESET();
      STATS_START(STATS_LOCAL);

            if (k != 0) {
```

```
                        start = MPI_Wtime();
                }
        for (int i= 1; i<=niter; i++) {

            // Perform the local smoothing on the RED Points
            ComputeLocal(*U,si,sj,RED,rhs);
            // Perform the local smoothing on the BLACK Points
            ComputeLocal(*U,si,sj,BLK,rhs);
        }
        STATS_STOP(STATS_LOCAL);
        timesLoc[k] = STATS_TIME(STATS_LOCAL);
        if (k != 0) {
           finish = MPI_Wtime();
         middle = finish - start + middle;
        }
    }

    ReportTimings(times,timesLoc , reps, chk_freq, niter, N,
                  si, sj, gi, gj);
      }
     catch (KelpErr & ke) {
         ke.abort();
         }

        return(middle);
}
```

Bibliography

[1] Alpern B., L. Carter, J. Ferrante, "Modeling Parallel Computers as Memory Hierarchies," *1993 Conference on Programming Models for Massively Parallel Computers*.

[2] Alpern, B., and L. Carter , "Towards a Model for Portable Performance: Exposing the Memory Hierarchy," 1992.

[3] Alpern, B., L. Carter, E. Feig, and T. Selker, "The Uniform Memory Hierarchy Model of Computation," IBM Watson Resarch Center, November 2, 1992.

[4] Ammon, J., "Hypercube connectivity with CC-NUMA architecture," SGI-Cray White paper, 1999.

[5] Anglano, C., J. Schopf, R. Wolski, and F. Berman, "Zoom: A Hierarchical Representation for Heterogeneous Applications," UCSD CSE Technical Report CS95-451, January 5, 1995.

[6] Argonne National Laboratories, "MPI - The Message Passing Interface Standard," `<http://www-unix.mcs.anl.gov/mpi/>`.

[7] Argonne National Laboratories, "MPICH-A Portable Implementation of MPI," `<http://www-unix.mcs.anl.gov/mpi/mpich/>`.

[8] Baden, S. B., "Tradeoffs in hierarchical algorithm design on multi-tier architectures."

[9] Baden, S. B., "Programming Abstractions for Dynamically Patitioning and Coordinating Localized Scientific Calculations Running on Multiprocessors," SIAM J. Sci. Stat. Comput., Vol. 12, No. 1, pp.145-157, January 1991.

[10] Baden, S. B., R. B. Frost, D. Shalit, "KeLP User Guide Version 1.3," Department of Computer Science and Engineering, University of California, San Diego, September 25, 1999.

[11] Baden, S. B. and S. J. Fink, "The Data Mover: A Machine-independent Abstraction for Managing Customized Data Motion," LCPC '99, August 1999.

[12] Baden, S. B. and S. J. Fink, "Communication overlap in multi-tier parallel algorithms," SC98, Orlando FL, Nov. 1998.

[13] Baden, S. B. and S. J. Fink, "A Programming Methodology for Dual-Tier Multicomputers," submitted for publication.

[14] Bader, David A. and Joseph Ja'Ja'. "SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs)," Tech. Rep. CS-TR-3798, Univ. of Maryland Inst. for Advanced Computer Studies-Dept. of Computer Sci, Univ. of Maryland, May 1997.

[15] Barney, Blaise M.. "POSIX Threads Programming," Maui High Performance Computing Center, October 29, 1998.

[16] Becker, D. J., T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A Parallel Workstation for Scientific Computation".

[17] Cappello, F. and O. Richard, "Performance characteristics of a network of commodity multiprocessors for the NAS benchmarks using a hybrid memory model," PACT 99, July 20, 1999.

[18] Carter, L., "Single Node Optimization," NPACI Parallel Computing Institute, August, 1998.

[19] Crandall, P. E. and M. J. Quinn, "A Decomposition Advisory System for Heterogeneous Data-Parallel Processing," Proceeding of the Third International Symposium on High Performance Distributed Computing.

[20] Crandall, P. E. and M. J. Quinn, "Non-Uniform 2-D Grid Partitioning for Heterogeneous Parallel Architectures," Proceedings of the 9th International Parallel Processing Symposium, 1995.

[21] de Supinski, B. R. and J. May, "Benchmarking Pthreads Performance," Lawrence Livermore National Labs, UCRL-JC-133263.

[22] Donaldson, S., J. M. D. Hill, and D. B. Skillicorn, "BSP Clusters: High Performance, Reliable and Very Low Cost," PRG-TR-5-98 Oxford University Computing Laboratory, 1998.

[23] Fink, S. J., "A Programming Model for Block-Structured Scientific Calculations on SMP Clusters," UCSD CSE Department/Ph.D Dissertation, June 1998.

[24] Fink, S. J., S. B. Baden, and S. R. Kohn, "Efficient Run-Time Support for Irregular Block-Structured Applications," Journal of Parallel and Distributed Computing, 1998.

[25] Fink, S.J., S.B. Baden, and S.R. Kohn, "Flexible Communication Mechanisms for Dynamic Structured Applications," IRREGULAR '96.

[26] Fink, S. J. and S. B. Baden, "Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters," ISCOPE '97, December 1997.

[27] Fink, S.J. and S.B. Baden, "Run-time Data Distribution for Block-Structured Applications on Distributed Memory Computers," CSE Technical Report Number CS94-386, September 1994.

[28] Foster, Ian, and Nicholas T. Karonis, "A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems(PS)," SC 98 Conference, Orlando FL, Nov. 1998.

[29] Gatlin, K.S. and L. Carter, "Architecture-Cognizant Divide and Conquer Algorithms," SC 99 Conference, Portland, OR, Nov. 1999.

[30] Kesselman, C., et al,
<http://www.globus.org/>.

[31] Gropp, W. W. and E. L. Lusk, "A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters," Proc. Programming Models for Massively Parallel Computers, October 1995, Berlin, pp. 2-7.

[32] Hill, J. M. D., P. I. Crumpton, D. A. Burgess, "The Theory, Practice and a Tool for BSP Performance Prediction Applied to a CFD Application," PRG-TR-4-1996 Oxford University Computing Laboratory, 1996.

[33] IBM PowerPC 604e User Manual, <http://www.chips.ibm.com/products/powerpc/chips/604e/604eUM_book.pdf>.

[34] Kleiman, S., D. Shah, B. Smaalders, "Programming with Threads," SunSoft Press.

[35] Kohn, S. R., "A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations," Ph.D dissertation, University of California at San Diego, La Jolla, CA, 1995.

[36] Lauden, J. and D. Lenowski., "The SGI Origin: A ccNUMA Highly Scalable Server," ISCA.

[37] Lawrence Livermore National Labs, "Using ASCI Blue Pacific,"
<http://www.llnl.gov/asci/platforms/bluepac/>.

[38] Lawrence Livermore National Labs, "The ASCI Program,"
<http://www.llnl.gov/asci/>.

[39] Grimshaw, A., et al,
<http://legion.virginia.edu/>.

[40] Lenoski, D., J. Lauden, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessor," Stanford University, March 1992.

[41] Lim, B.-H., P. Heidelberger, P. Pattnaik, and M. Snir, "Message Proxies for Efficient, Protected Communication on SMP Clusters," Proc. Third Int'l Symp. on High-Performance Computer Architecture, San Antonio, TX, Feb. 1997, IEEE Computer Society Press, pp. 116-27.

[42] Lumetta, S. S., A. M. Mainwaring, and D. E. Culler, "Multi-protocol active messages on a cluster of SMPS," in Proc. SC97, Nov. 1997.

[43] Majumdar, A., "Parallel Monte Carlo Photon Transport," NPACI Parallel Computing Training, 1999.

[44] May, J., B. de Supinski, B. Pudliner, S. Taylor, and S. Baden, "Final Report Programming Models for Shared Memory Clusters," Lawrence Livermore National Labs, 99-ERD-009, January 13, 2000.

[45] May, J. and B. R. de Supinski, "Experience with Mixed MPI/Threaded Programming Models," Lawrence Livermore National Labs, UCRL-JC-133213.

[46] Mitchell, N., L. Carter, J. Ferrante, and K. Högstedt, "Quantifying the Multi-Level Nature of Tiling Interactions," LCPC 1997.

[47] National Center for Supercomputing Applications (NCSA) at University of Illinois, Urbana-Champaign; part of The Alliance, <http://www.ncsa.uiuc.edu/>, <http://www.uiuc.edu/>, <http://www.ncsa.edu/>.

[48] NCSA Silicon Graphics Origin2000,
<http://www.ncsa.uiuc.edu/SCD/Hardware/Origin2000/>.

[49] Nguyen, T., M. M. Strout, L. Carter, and J. Ferrante, "Asynchronous Dynamic Load Balancing of Tiles," SIAM 99.

[50] OpenMP Architecture Review Board, OpenMP FAQ,
<http://www.openmp.org/index.cgi?faq>.

[51] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface 1.0," Oct. 1997.

[52] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface 1.0," Oct. 1998.

[53] Patterson and Hennessy, Computer Architecture: A Quantitative Approach, 2nd Ed., Morgan Kaufmann.

[54] Peisert, S., S. Mock, and S. Baden, "Simulating Neurotransmitter Activity on Parallel Processors," UCSD Graduate Research Conference, 1999.

[55] Pfister, G. F., "In Search of Clusters - The Coming Battle in Lowly Parallel Computing," Prentice Hall PTR, 1995.

[56] Pilkington, J. R. and S. B. Baden, "Partitioning with Spacefilling Curves," CSE Technical Report Number CS94-349, March 1994.

[57] Pilkington, J. R. and S. B. Baden, "Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves," January 10, 1995.

[58] Baden, S. B. "RedBlack-3D," 1999.

[59] Ridge, D., B. Becker, P. Merkey, and T. Sterling, "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs."

[60] Schopf, J. M. and F. Berman, "Performance Prediction Using Intervals," UCSD CSE Dept Technical Report CS97-541, May 1997.

[61] Schopf, J. M. and F. Berman, "Performance Prediction in Production Environments," UCSD CSE Dept. Technical Report CS97-558, September 1997.

[62] San Diego Supercomputer Center (SDSC) <http://www.sdsc.edu/>.

[63] SGI-Cray, "Models of Parallel Computation."

[64] SGI-Cray, "Origin2000 and Onyx2 Performance Tuning and Optimization Guide (IRIX 6.5)," <http://techpubs.sgi.com/library/tpl/cgi-bin/browse.cgi?coll=0650&db=bks&cmd=toc&pth=/SGI_Developer/OrOn2_PfTune>.

[65] Simon, H. D. and S. H. Teng, "How Good is Recursive Bisection?" SIAM J. S. C., 1995.

[66] Smallen, S., W. Cirne, J. Frey, F. Berman, R. Wolski, M-H. Su, C. Kesselman, S. Young, M. Ellisman, "Combining Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience," October 12, 1999.

[67] Sun Microsystems, Inc., "Sun Servers," <http://www.sun.com/servers/>.

[68] Sun Microsystems, Inc., "UltraSPARC-II Products," <http://www.sun.com/microelectronics/UltraSPARC-II/index.html>.

[69] Wolski, R., N. Spring, and C. Peterson, "Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service," Proceedings of Supercomputing 1997.