AFIT/GE/ENG/99M-02

ASYMMETRIC LOAD BALANCING ON A HETEROGENEOUS CLUSTER OF PCs

THESIS

Christopher A. Bohn
Captain, USAF

AFIT/GE/ENG/99M-02

19990413 101

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 1999	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Asymmetric Load Balancing on a Heterogeneous Cluster of PCs	**5. FUNDING NUMBERS**
6. AUTHOR(S) Christopher A. Bohn, Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2950 P Street Wright-Patterson AFB OH 45433-7126	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GE/ENG/99M-02
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Jeff E. Graham, ASC MSRC PET Director ASC/HP 2435 Fifth St WPAFB, OH 45433-7802 COMM: (937)255-3995x231 DSN: 785-3995x231 EMAIL: jeff.graham@msrc.wpafb.af.mil	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
Advisor: Gary B. Lamont
COMM: (937)255-3636x4718 DSN: 785-3636x4718
EMAIL: gary.lamont@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

In recent years, high performance computing with commodity clusters of personal computers has become an active area of research. Many organizations build them because they need the computational speedup provided by parallel processing but cannot afford to purchase a supercomputer. With commercial supercomputers and homogenous clusters of PCs, applications that can be statically load balanced are done so by assigning equal tasks to each processor. With heterogeneous clusters, the system designers have the option of quickly adding newer hardware that is more powerful than the existing hardware. When this is done, the assignment of equal tasks to each processor results in suboptimal performance. This research addresses techniques by which the size of the tasks assigned to processors is a suitable match to the processors themselves, in which the more powerful processors can do more work, and the less powerful processors perform less work. We find that when the range of processing power is narrow, some benefit can be achieved with asymmetric load balancing. When the range of processing power is broad, dramatic improvements in performance are realized – our experiments have shown up to 92% improvement when asymmetrically load balancing a modified version of the NAS Parallel Benchmarks' LU application.

14. SUBJECT TERMS Parallel Processing, Pile of PCs, Heterogeneous Cluster, Beowulf, Network of Workstations, Load Balancing	**15. NUMBER OF PAGES** 201
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

AFIT/GE/ENG/99M-02

ASYMMETRIC LOAD BALANCING ON A HETEROGENEOUS CLUSTER OF PCs

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Christopher A. Bohn, B.S.E.E., M.S.

Captain, USAF

March 1999

AFIT/GE/ENG/99M-02

ASYMMETRIC LOAD BALANCING ON A HETEROGENEOUS CLUSTER OF PCs

Christopher A. Bohn, B.S.E.E., M.S.
Captain, USAF

Approved:

[illegible] — 23 FEB 1999
Gary B. Lamont, Ph.D.
Chairman — date

[illegible] — 23 Feb 1999
Jeffrey K. Little, Ph.D., Lt Col, USAF
Member — date

[illegible] — 23 Feb 99
Richard A. Raines, Ph.D., Maj, USAF
Member — date

Acknowledgements

I really must first thank my beautiful wife, Caryl, for her caring and understanding. She kept me sane for the last eighteen months. I could not have done this without her. Caryl, I love you. My border collie, Oreo, deserves a nod for reminding me that occasionally you just have to drop what you're doing and run wildly around the room.

I'd also like to thank my advisor, Dr Gary Lamont, and my committee members, Lt Col Jeffrey Little and Maj Richard Raines. Their patience and guidance have been instrumental, particularly early in my thesis effort.

I'd also like to thank Capt Lonnie Hammack and Capt Fernando Silva for conducting the "ping-pong" tests that characterized the ABC's network throughput. These two and Capt Karl Deerman warrant another "thank you" for acting as sounding-boards for some of my ideas and providing suggestions for me to consider.

Finally, I'd like to thank the US Air Force for giving me this opportunity.

Christopher A. Bohn

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ABC	AFIT Bimodal Cluster
AFIT	Air Force Institute of Technology
AFRL	Air Force Research Laboratory
AMD	Advanced Micro Devices
ASCI	Accelerated Strategic Computing Initiative
ASCII	American Standard Code for Information Interchange
BST	Binary Search Tree
CCOTS	Commodity-Commercial-off-the-Shelf
CFD	Computational Fluid Dynamics
COTS	Commercial-off-the-Shelf
COW	Cluster of Workstations
CRCW	Concurrent-Read, Concurrent-Write
CREW	Concurrent-Read, Exclusive-Write
CTA	Computational Technology Area
DCPC	Dedicated Cluster Parallel Computer
DEC	Digital Equipment Corporation
DoD	Department of Defense
DoE	Department of Energy
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared-Memory
ERCW	Exclusive-Read, Concurrent-Write
EREW	Exclusive-Read, Exclusive-Write
FDM	Finite Difference Method
FFT	Fast Fourier Transform
flops	Floating Point Operations per Second
FPU	Floating Point Unit
FSF	Free Software Foundation
GNU	GNU's not UNIX
HP	Hewlett-Packard
HPCM	High Performance Computing Modernization
IA	Intel Architecture instruction set
IBM	International Business Machines
ICN	Interconnection Network
ILP	Instruction-Level Parallelism
LANL	Los Alamos National Laboratory
MCSE	Microsoft Certified System Engineer
MIMD	Multiple Instruction Stream-Multiple Data Stream
MIPS	Million Instructions per Second
MISD	Multiple Instruction Stream-Single Data Stream

MPI	Message Passing Interface
MPP	Massively Parallel Processors
NaN	Not-a-Number
NAS	National Aerospace Simulation Facility
NASA	National Aeronautics and Space Administration
NFS	Network File System
NIS	Network Information Service
NORMA	No Remote Memory Access
NOW	Network of Workstations
NPB	NAS Parallel Benchmarks
NSF	National Science Foundation
NUMA	Non-Uniform Memory Access
OS	Operating System
PC	Personal Computer
PoPC	Pile of PCs
PDE	Partial Differential Equation
PRAM	Parallel Random Access Machine
PVM	Parallel Virtual Machine
QUIPS	Quality Improvements per Second
RAM	Random Access Machine
RAM	Random Access Memory
SGI	Silicon Graphics, Incorporated
SIMD	Single Instruction Stream-Multiple Data Stream
SISD	Single Instruction Stream-Single Data Stream
SMP	Symmetric Multiprocessor
SPMD	Single Program-Multiple Data
SDRAM	Synchronous DRAM
SRAM	Static Random Access Memory
SSOR	Successive Symmetric Over-Relaxation
UMA	Uniform Memory Access
UML	Unified Modeling Language
UNITY	Unbounded Nondeterministic Iterative Transformations
US	United States
VLIW	Very Large Instruction Word
VM	Virtual Memory

THIS PAGE INTENTIONALLY LEFT BLANK

Abstract

In recent years, high performance computing with commodity clusters of personal computers has become an active area of research. Many organizations build them because they need the computational speedup provided by parallel processing but cannot afford to purchase a supercomputer.

With commercial supercomputers and homogenous clusters of PCs, applications that can be statically load balanced are done so by assigning equal tasks to each processor. With heterogeneous clusters, the system designers have the option of quickly adding newer hardware that is more powerful than the existing hardware. When this is done, the assignment of equal tasks to each processor results in suboptimal performance.

This research addresses techniques by which the size of the tasks assigned to processors is a suitable match to the processors themselves, in which the more powerful processors can do more work, and the less powerful processors perform less work. We find that when the range of processing power is narrow, some benefit can be achieved with asymmetric load balancing. When the range of processing power is broad, dramatic improvements in performance are realized – our experiments have shown up to 92% improvement when asymmetrically load balancing a modified version of the NAS Parallel Benchmarks' LU application.

THIS PAGE INTENTIONALLY LEFT BLANK

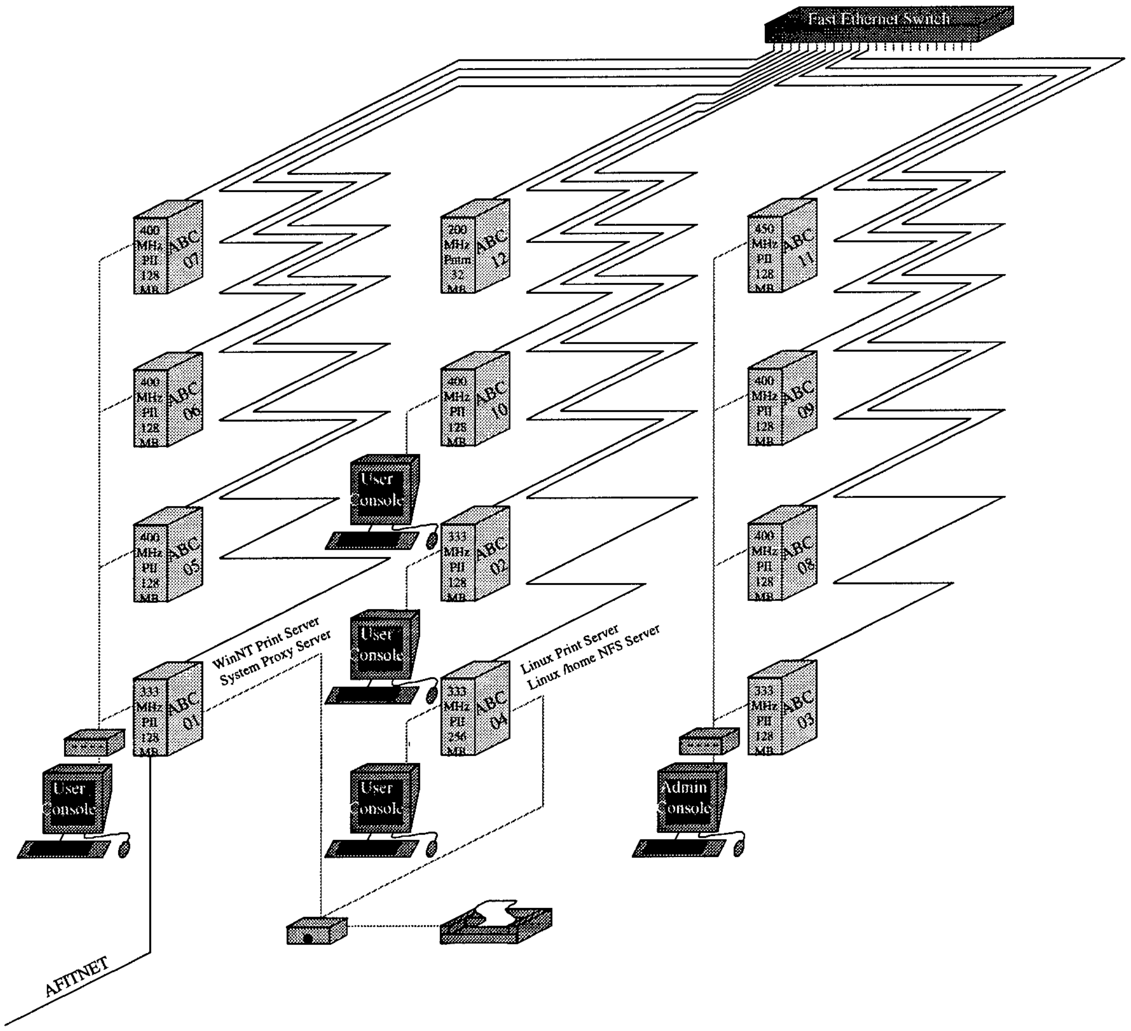
I. Introduction

Traditionally, supercomputers are designed with the objective of achieving the greatest computational performance physically possible; the the U.S. Department of Energy's (DoE) Accelerated Strategic Computing Initiative (ASCI) [23] is the current embodiment of this niche. At the other extreme has been low-cost computer designs, where the performance is subordinate to the end-user price; commodity personal computers (PC) traditionally filled this role. Between the two lay the designs that focus on the price/performance ratio, exemplified by scientific workstations [35:17]. Advances in the performance of commodity PCs and commodity networks without corresponding increases in price led to the discovery that supercomputing performance can be realized with clusters of PCs, at a price/performance ratio an order of magnitude better than is possible with typical supercomputers [85]. The AFIT Bimodal Cluster (ABC) is one such system.

1.1. The AFIT Bimodal Cluster

In the spring of 1998, motivated by AFIT's previous experience with networks of workstations (NOWs) [31][102] and by the Beowulf Project [53][72][84][85][94], a group of AFIT students under the direction of Professor Gary Lamont began construction on the AFIT Bimodal Cluster for computer architectural research. Such an effort indirectly supports the US National Science Foundation (NSF) Grand Challenge problems [60] and the Department of Defense High Performance Computing Modernization (DoD HPCM) Computational Technology Areas (CTAs) [38].

The ABC is a "Pile of Personal Computers" (PoPC) that operates under both Windows NT and Linux operating systems (OS) and is intended to evolve over time as additional hardware and software became available. The inaugural parallel code ran on the evening of 19 May 1998 using four 333 MHz Intel Pentium II uniprocessor nodes interconnected with a 100BaseT Fast Ethernet hub.



Fast Ethernet Switch
400 MHz PII 128 MB ABC 07
200 MHz Pntm 32 MB ABC 12
450 MHz PII 128 MB ABC 11
400 MHz PII 128 MB ABC 06
400 MHz PII 128 MB ABC 10
400 MHz PII 128 MB ABC 09
User Console
400 MHz PII 128 MB ABC 05
333 MHz PII 128 MB ABC 02
400 MHz PII 128 MB ABC 08
WinNT Print Server
System Proxy Server
User Console
Linux Print Server
Linux /home NFS Server
333 MHz PII 128 MB ABC 01
333 MHz PII 256 MB ABC 04
333 MHz PII 128 MB ABC 03
User Console
User Console
Admin Console
AFITNET

Figure 1-1. The AFIT Bimodal Cluster, as of 5 Jan 99.

An early decision was that future expansion of the ABC would not be limited due to previous design decisions; as such, the ABC's hardware would be heterogeneous.[1] Since the project was begun, the ABC has grown to twelve nodes, including the original four, six 400 MHz Pentium IIs, a 450 MHz Pentium II, and a 200 MHz Pentium; further, the Fast Ethernet hub has been replaced with a Fast Ethernet switch (Figure 1-1). A detailed description and development discussion of the ABC can be found in Sections 2.2 and 3.1.

A consequence of this decision is that the performance realized by newer hardware would be limited by the performance the older hardware could offer. If workloads were matched to the processors' capabilities, then this limitation would be overcome and the older hardware would continue to be able to contribute to the solution of computational challenges. In this fashion, obsolescence of older technologies would be delayed, further reducing the cost of high performance computing.

1.2. Research Overview

1.2.1. Rationale

One field of study that includes both Grand Challenge and CTA efforts is computational fluid dynamics (CFD) [5][36]. CFD has a number of research priorities that include parallel processing and turbulence modeling [2]. In particular, the CFD investigation of turbulence induced by surface roughness in high-speed airflows using both

[1] "Heterogeneous" has different connotations, ranging from different underlying architectures to different user loads. In the context of this thesis, the ABC is heterogeneous in that the nodes have processors clocked at different rates, that the nodes have different implementations of the Intel Architecture instruction set (IA), and that the memories are different sizes and are clocked at different rates. The ABC is also heterogeneous in that two distinct operating systems are used, though only Linux is within the scope of this thesis effort.

windtunnels and CFD modeling is a current area of research at AFIT in the Department of Aeronautics and Astronautics. These researchers are addressing the critical high-speed CFD problem where existing CFD high-speed turbulence models are incorrect. This is because the nondeterministic nature of turbulence has led to time-averaged analysis, rather than instantaneous analysis, and further, the data for high-speed turbulence generally is extrapolated from low-speed incompressible turbulence models [49:1-6].

1.2.2. Objectives

The research described in this document is not intended to advocate the use of PoPCs *en lieu* of commercial supercomputers. Rather, given that commodity clusters of PCs do exist, we address the issue of how to make more efficient use of these clusters. As a point of comparison, though, we do cite some reported results of our test application on other platforms.

Likewise, this thesis effort does not address any new mathematical modeling techniques in the CFD realm, but rather supports CFD research by establishing computational techniques to make more optimal use of a PoPC. In particular, this effort focuses on optimizing a specific CFD application on the ABC and on heterogeneous parallel architectures in general.

In the interest of focusing on the parallel architecture problem and not on the fluid dynamics problem, we make use of a well-known CFD benchmark [99][10], described in Section 2.3. The use of a CFD problem domain is entirely appropriate, as the technique used for computational fluid dynamics is applicable to other problem domains as well. These other problem domains not only include problems solved similarly to CFD, such as computational electromagnetics [3][37], but also any data-decomposed supercomputing problem. The benchmark used in this thesis effort is appropriate since it is designed specifically to mimic the computation and communication patterns of computational fluid dynamics applications [75:2].

Thus, the specific objectives of this research are:

a) Develop an algorithm to modify the static partitioning of a data-decomposed parallel application at run-time from a symmetric decomposition to an asymmetric decomposition;
b) Develop techniques to measure the relative computational capabilities of the nodes in a heterogeneous cluster of PCs;
c) Incorporate the algorithm and measurement software into a CFD application;
d) Provide a statistical analysis of the resulting performance and a comparison with other platforms.

As a result of this research, future computational scientists should be able to take advantage of the capabilities of the newest technologies while still using older technologies. This, in turn, delays the obsolescence of equipment in a field where capabilities double every eighteen months.[2]

1.2.3. Approach

The first step in this journey was the construction of the ABC PoPC. The author of this document undertook the responsibility of the physical construction, and he was

[2] Generally speaking, Moore's Law is invoked when expressing the fact that computer systems improve at an exponential rate. More specifically, Moore's Law states that the logic capacity of silicon doubles about every 18 months, and the law has often been extended to include microprocessor performance [40][67].

assisted by the other students in the project. The author also assumed responsibility as the Linux system administrator, learning this role along the way.

The next task was the selection of the CFD application on which to test the heterogeneous load balancing algorithm, detailed in Section 3.2. Studying the design of the application was necessary to understand the assumptions implicit in its encoding and how its symmetric data decomposition is defined. After this, we designed and implemented the necessary changes to the application to permit asymmetric load balancing. Concurrent with the development of the load balancing algorithm was the design and implementation of a library that the load balancing algorithm uses to assess the capabilities of the compute nodes. We then designed and conducted experiments to test the modifications and statistically assessed the results to determine if and how much the changes improve the performance of the application on a heterogeneous cluster.

1.3. Document Overview

The remainder of this document is organized thus:

Chapter II provides the background necessary to understand this thesis effort. This begins with a discussion on commodity supercomputers and the factors that led to them. This discussion then leads into a more detailed description of the system used for this thesis effort. Next, the application that was modified for the experiments is described. Finally, a discussion on load balancing is offered, including an analogy to convey the concept of load balancing, previous load balancing efforts, and why load balancing is important in this case. Supplemental background material can be found in Appendix A.

Chapter `III` details the approach used in this enterprise. The chapter begins by explaining how we selected the experimental application, and describes the process by which the application was modified to implement asymmetric load balancing. Next, the design and implementation of the library that provides the load balancing algorithm with the necessary information is outlined. Finally, we discuss how we tested the load balancing techniques.

Chapter `IV` provides the results and analysis of those tests. The performance of the application in the major tests is examined, as is the improvement over the non-load balanced performance. The scalability of the application on the ABC, both before and after load balancing, is also addressed. Tables of the raw data are available in Appendix D.

Chapter `V` offers conclusions about asymmetric load balancing and the different weighting approaches tested. A discussion on future directions is also provided for both asymmetric load balancing research and for the growth of the AFIT Bimodal Cluster.

Appendix A provides additional background material on data partitioning approaches and the finite difference method of solving systems of partial differential equations (PDE), that is not vitally necessary to understand this document but may help the interested reader who is unfamiliar with concepts tangential to this thesis effort.

Appendix B lists the "diff" files for the application used in the experiments. Full listings of the source code is impractical; however, the UNIX `diff` command [30:2–34 to 2–35] permits a listing of only the changes between the original code and the modified code. The `patch` command [93] can then be used to reconstruct the new code from the original,[3] or vice-versa. Here, the list of changes allows the readership to study the code implementing the load balancing schemes described in Sections 3.3 and A.1.

Appendix C lists the source code that implements the design in Section 3.3.2.2 to measure the relative capabilities of the compute nodes.

Appendix D is a repository for tables of the results of the experiments. Included are the performance values for each of the experiments, as well as the data partition sizes for each of the experiments.

Throughout this document, the assumptions about readership are:

a) Understanding of computer architecture.

b) Understanding of algorithms.

c) Familiarity with basic parallel & distributed programming concepts.

[3] Available from [99].

II. Background

This chapter provides the reader with the appropriate background to understand the necessity, approach, and results of this thesis effort. A description of commodity supercomputing is provided, along with explanations of why commodity supercomputing has become an important area of research. Next, material directly relevant to this thesis investigation is described: a description of the system used for the experiments, an outline of the NAS Parallel Benchmarks [99], which includes the application modified for the experiments, and an explanation of load balancing and why it can offer a dramatic performance improvement on a heterogeneous platform. The material presented does not include computer architecture [64][35], algorithms [16], basic parallel & distributed processing concepts [48][4], fluid dynamics [42], or computational fluid dynamics [7]; the reader who is unfamiliar with a concept may find explanations in Appendix A or in the references.

2.1. Commodity Supercomputing

Massively parallel processor (MPP) machines are those systems designed for very-high-end applications that demand the highest computational and interprocessor communcation capabilities. An MPP uses commodity processors on the nodes, interconnected by a high-bandwidth, low-latency network. MPPs can be scaled up to hundreds of nodes, and MPPs with thousands of nodes are not unheard-of [43:28].

While there is clearly a continuing demand for MPPs, they suffer from weaknesses that are not shared by new classes of supercomputers. For example, an MPP design takes up to two additional years of engineering effort than is required to develop desktop workstations from the same components [8:55]. At the current rate of performance increase, this yields performances about half those possible if "just-in-time" configuration were possible. This extra engineering effort (and extra development costs) is not only in the hardware design, but also due to a parasitic redesign of the OS and other software – the system software developed for a workstation using a certain processor is suboptimal for an MPP node, and drivers for the unique hardware configurations must be thoroughly tested [8:55-56][94].

Ten years ago, Gordon Bell predicted that the diseconomy of scale for supercomputers would lead to only the largest applications getting executed on systems with the most computational power. More and more challenging applications being investigated by budget-restricted researchers would be performed on distributed lower-end computers working in concert [12:1094-1095,1100]. Five years ago, technological advances resulted in the initiation of two projects that would bring supercomputing capabilities to researchers on budgets. They were the Berkeley NOW Project [8] and the NASA Gigaflops Workstation Project[4] [84].

2.1.1. Networks of Workstations

While parallel computing on clusters of workstations (COWs) using commercial-off-the-shelf (COTS) equipment has been around since 1991 [8:56], advances in processor and network technology led a team at the University of California at Berkeley to undertake a massive Network of Workstations (NOW) project in 1994 with the overall

[4] Often referred to as "The Beowulf Project."

objective of making a system comparable in performance to supercomputers at that time. The specific objectives of the investigation were:

a) Use of the aggregate DRAM among the workstations as backing store for virtual memory (VM), in lieu of using a hard disk;
b) Allowing workstations access to each others' file caches;
c) Use of the aggregate disk space among the workstations as a redundant array of independent disks (RAID);
d) Development of a low-overhead, low-latency communication library to replace the Parallel Virtual Machine (PVM) library;
e) The impact that local sequential jobs and spawned parallel processes have on each other when workstations are available for both interactive use and supercomputing; and
f) A robust global operating system for the NOW, built on top of the native OS, providing a "guarantee" of stand-alone workstation performance or better to every user

[8:56-62].

In 1996, to study the utility of a COTS NOW in meeting the US Air Force's high performance computing needs, particularly in the field of digital signal and image processing, students at AFIT constructed a COW consisting of five Sun Ultra Sparc 1s and an Ultra Sparc 2, networked by 10BaseT switched Ethernet or by Myrinet [31][102].

2.1.2. Beowulf-Class Supercomputers

At the same time the Berkeley NOW Team began its investigations, the Earth and Space Science division at NASA's Goddard Space Flight Center initiated the Gigaflops Workstation Project with the mandate of developing a "Gigaflops Scientific Workstation" costing no greater than $50,000, which was then the price of a high-end scientific workstation. The architects of the prototypical system, "Beowulf,"[5] kept the price under $50,000 by using only commodity components[6] and open-source, free-license software[7] that allowed optimization of the OS (Linux) for the architecture and application, though it achieved only 60 Mflops [84]. By 1996, though, the combined benefits of more powerful commodity processors, less expensive high-speed networks, and free software permitted Beowulf-class system constructed from sixteen Intel Pentium Pro machines networked by dual 100BaseT switched Fast Ethernets to sustain 1.25Gflops for $50,000 [72].

In the years since Beowulf was demonstrated, government research laboratories, academic institutions, and commercial vendors throughout the world have constructed Beowulf-class systems and PoPC's,[8] taking advantage of the very low price afforded by the economies of scale available from commodity PCs & networks and from free-license software. While most commonly implemented with Intel x86 processors, many

[5] There is no particular significance to associating the name with the Beowulf legend, other than "it just sounded cool" [13].

[6] Sixteen Intel 80486DX4-based personal computers interconnected with 10BaseT Ethernet and 10Base2 channel-bonded Ethernet.

[7] The issue of free vs. proprietary software and open-source vs. closed-source are beyond the scope of this thesis (as is the debate over "free software" vs. "open-source software"), except to emphasize that free software permits customization of the OS and device drivers, reduces the expense of building a large system, and uses a development model that assures rapid identification and correction of bugs. The interested reader should see [68][69][90].

[8] The exact definition of a Beowulf is a subject of some debate [13][66]. "PoPC" is a more general system description than "Beowulf," and does not specify that a single-system image be maintained, nor does a free operating system need be used [50].

Beowulves are constructed with DEC Alpha processors [13]. Less commonly implemented, though occasionally discussed are systems using IBM/Motorola PowerPC, Sun SPARC, Motorola 68k, and Acorn ARM processors [52].

Trying to determine precisely how many such systems exist is not easy since most are constructed by the researchers themselves, rather than purchased fully assembled from supercomputer vendors. There are indicators, however. The union of three websites [24][52][71] and [28][47][79][73][87] indicate there are at least 78 clusters at 60 sites using Linux, Solaris, and Windows NT.[9] There are at least three commercial vendors of high performance clusters, Alta Technology [6], DCG Computers [21], and Paralogic [63]. Finally, the Beowulf Mailing List [13] has a total of 762 subscribers from 644 internet domains [55]. Examining several sources [24][28][52][55][79][71][73][87] reveals there are at least nineteen countries with high performance clusters.[10] Some of the more notable systems are listed in Table 2-1.

Beowulf defines a genre of supercomputers known for their price-to-performance ratios. In 1997, the Gordon Bell Prize for Price/Performance was awarded to a 32-processor Pentium Pro-based Beowulf[11] that sustained $47/Mflop on an *n*-body treecode [95]. More recently, a 70-processor DEC Alpha-based Beowulf, DoE's "Avalon," took second-place in the 1998 Gordon Bell Prize in the same category after sustaining

[9] Does not include "enterprise clusters" designed to provide fail-over and similar high-reliability services.
[10] Countries known to have PoPCs: Australia, Belgium, Brazil, Canada, Czech Republic, France, Germany, India, Iran, Israel, Italy, Japan, Spain, Sweden, Switzerland, Taiwan, Thailand, the United Kingdom, and the United States.
[11] DoE's 16-processor "Loki" and the California Institute of Technology's 16-processor "Hyglac" networked together.

$15/Mflop on molecular dynamics code, losing out to an application-specific computer [96][41].

Table 2-1. Noteworthy PC Clusters.

Name Location URL	Processors Network	Significance
PAPERS portable demonstrator Purdue University http://garage.ecn.purdue.edu/~papers	4 80486 PAPERS (experimental custom network)	20-pound portable cluster
Stone SouperComputer Oak Ridge National Laboratory http://www.esd.ornl.gov/facilities/beowulf/	126 (mostly 80486; some Pentium) Ethernet	All nodes are "surplus" desktop computers. "Zero dollars per node."
Megalon Sandia National Laboratory http://megalon.ca.sandia.gov/	56 Pentium Pro Fast Ethernet	14 nodes, each 4-way SMP
theHive Goddard Space Flight Center http://newton.gsfc.nasa.gov/thehive/	128 Pentium Pro Fast Ethernet	64 nodes, each 2-way SMP. First to exceed 100 processors.
Avalon Los Alamos National Laboratory http://cnls.lanl.gov/avalon/	140 Alpha 21164 Fast Ethernet with Gb Ethernet cross-links	First on Top500 list. Currently #113 on Top500 list.
C-Plant Sandia National Laboratory http://www.cs.sandia.gov/cplant/	400 Alpha 21164 Myrinet	Currently #97 on Top500 list.
NT SuperCluster Univeristy of Illinois at Champaign-Urbana http://www-esag.cs.uiuc.edu/projects/clusters.html	256 Pentium II Myrinet	128 nodes, each 2-way SMP. Large-Scale Windows NT cluster.
CLOWN University of Paderborn http://www.linux-magazin.de/cluster/index.en.html http://www.heise.de/ix/artikel/E/1999/01/010/	512 x86 (Pentium, Pentium Pro, Pentium II) and 60 Alpha 21x64. Fast Ethernet with Gb Ethernet cross-links	Assembled in 12 hours. Executed "real-world" code and benchmarks, and disassembled same weekend (5-6 Dec 98).

While Beowulf designs give researchers "fantastic" price/performance ratios, they also can bring high performance as well. In June 1988, the supercomputing community observed that Beowulves can compete with traditional supercomputers in terms of raw performance: the judges of the Top500[12] list ranked Avalon as the 315th most powerful supercomputer in the world due to its 19.2 Gflops performance on the parallel LINPACK benchmark [25]. Since then, DoE doubled the number of Avalon's processors and added more memory to the existing nodes, bringing its LINPACK performance to 48.6 Gflops and outperforming all but 112 of the world's supercomputers. Meanwhile, another DoE cluster of commodity Alpha PCs interconnected with the proprietary Myrinet [59] network achieved 54.2 Gflops and was ranked number 97 on the November 1998 Top500 list [26].

To investigate the usefulness of PoPC's for DoD applications, students in AFIT's parallel & distributed computing laboratory began work in 1998 on the AFIT Bimodal Cluster.

2.2. *The AFIT Bimodal Cluster – System Description*

The ABC is a continuously-evolving PoPC built with the just-in-time approach to hardware configuration. It differs from a Beowulf-class supercomputer in that

a) it hosts a proprietary, closed-source operating system (Windows NT) in addition to a free-license, open-source operating system (Linux), and
b) a single-system image is not maintained.

The ABC can be booted under either of two operating systems. All compute nodes but one, have Microsoft Windows NT[13] 4.0 (SP4) [56] and Linux 2.0.33 [70] (with Beowulf [54] enhancements) installed and configured for cluster computing; the remaining

[12] A list compiled twice each year of the 500 most powerful supercomputers, as ranked by the LINPACK benchmark [89].
[13] Three with Windows NT Server, and all others with Windows NT Workstation

node has only Linux installed. This last node is a nineteen-month-old[14] personal computer already at AFIT that was donated to the project; since we have not yet purchased a Windows NT license for it, it has only Linux installed.

Because just-in-time configuration is used, the capabilities of each node are, in general, different from the other nodes. In its current configuration (Figure 1-1), the ABC is built from twelve uniprocessor nodes interconnected by a 100 Mbps Fast Ethernet switch. One of these nodes uses a 200 MHz Intel Pentium[15] processor; four use 333 MHz Intel Pentium II[16] processors; six use 400 MHz Pentium II processors; and one uses a 450 MHz Pentium II processor.

The ABC's interconnection network is 100 Mbps Full-Duplex Fast Ethernet, using an Intel Express 510T switch. The 510T's switching fabric has an internal capacity of 6.3 Gbps, providing an effective aggregate network capacity of 800 Mbps [45:78].

The memory configuration of the ABC is as diverse as the processor configuration. One node has 32 MB 15 ns DRAM, three have 128 MB 15 ns SDRAM, one has 256 MB 15 ns SDRAM, and seven have 128 MB 10 ns SDRAM. This gives it an aggregate 1.53 GB of distributed memory.

Several tools are available; for this research, the Free Software Foundation (FSF) GNU egcs 1.0.2 compiler suite [19], particularly the egcs implementations of gcc and

[14] According to AFIT/SC's records, the computer was purchased in August 1997; it was added to the ABC seventeen months later in January 1999. At the time of publication, two additional months have passed.
[15] For a description of the Pentium design, see [14:679-696].
[16] For a detailed description of the Pentium II design, see [58].

g77,[17] are used with the MPICH 1.1.0 [9] implementation of MPI. Details on which processors were used can be found in Section 3.5.

Table 2-2. Characteristics of ABC Nodes.

Node	Date Installed	Processor	Memory	Operating System
ABC01	April 1998	333 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Server Linux 2.0.33
ABC02	April 1998	333 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Server Linux 2.0.33
ABC03	April 1998	333 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC04	April 1998	333 MHz Pentium II	256 MB SDRAM 256 MB swapspace	Windows NT 4.0 Server Linux 2.0.33
ABC05	August 1998	400 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC06	August 1998	400 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC07	August 1998	400 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC08	August 1998	400 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC09	August 1998	400 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC10	August 1998	400 MHz Pentium II	128 MB SDRAM 128 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC11	December 1998	450 MHz Pentium II	128 MB SDRAM 256 MB swapspace	Windows NT 4.0 Workstation Linux 2.0.33
ABC12	January 1999	200 MHz Pentium	32 MB DRAM 64 MB swapspace	Linux 2.0.33

2.3. NAS Parallel Benchmarks

The original NAS Parallel Benchmarks (NPB) were developed at NASA Ames Research Center in 1991 as problem specifications, which researchers and supercomputer vendors could then implement. The five kernels (Table 2-3) and three simulated CFD applications (Table 2-4) in NPB were intended to allow demonstrations of systems' suitability for aerophysics applications. Each of the kernels focused on a particular type of

[17] -O3 optimization for C and –O optimization for Fortran.

numerical computation, while the simulated applications were specified such that they represent data structures, data movement, and computational techniques that are typically found in real CFD applications [75:2].

Table 2-3. NAS Parallel Benchmarks – Kernels.

Benchmark Name	Abb.	Description
Embarassingly Parallel	EP	Accumulate 2D statistics of large number of pseudorandom numbers
Multigrid	MG	Solved 3D Poisson PDE, with constant coefficients
Congjugate Gradient	CG	Computes approximation to smallest eigenvalue of large, sparse matrix
FFT PDE	FT	Solves 3D PDE using FFTs
Integer Sort	IS	Sorts array of integers

[75:15-16]

Table 2-4. NAS Parallel Benchmarks – Simulated CFD Applications.

Benchmark Name	Abb.	Description
Lower-Upper Diagonal	LU	Uses symmetric successive over-relaxation (SSOR) to solve regular-sparse, block 5x5 lower & upper triangular system of equations that are the product of unfactored implicit finite-difference discretization of three-dimensional Navier-Stokes equations
Scalar Pentadiagonal	SP	Solves multiple independent systems of nondiagonally-dominant, scalar pentadiagonal equations resulting from approximately-factored implicit finite-difference discretization of Navier-Stokes equations
Block Tridiagonal	BT	Solves multiple independent systems of nondiagonally-dominant, block 5x5 tridiagonal equations resulting from approximately-factored implicit finite-difference discretization of Navier-Stokes equations

[75:16][10:5]

By 1995, some shortcomings of NPB 1 benchmarks led to the development of the NPB 2 benchmarks. These shortcomings included [10:3]:

a) The implementations tended to be tuned to the particular system by vendors. While these implementations demonstrated what the specific system is capable of doing, they were not representative of the performance which a typical computational scientist/engineer could expect for a specific application.

b) The vendor-implemented software generally was also proprietary, preventing researchers from using the vendors' techniques to obtain better performance.

c) The system-specific implementations were not very portable due to compiler/assembly language tuning.

d) The largest problem size specified in NPB 1 was no longer representative of the largest real-world problems.

Table 2-5. NPB Problem Sizes.

Benchmark	Class S "sample"	Class W "workstation"	Class A	Class B	Class C
EP	2^{24}	2^{25}	2^{28}	2^{30}	2^{32}
MG	32^3	64^3	256^3	256^3	512^3
CG	1,400	7,000	14,000	75,000	150,000
FT	64^3	128^2x32	256^2x128	512x256^2	512^3
IS	2^{16}	2^{20}	2^{23}	2^{25}	2^{27}
LU	12^3	33^3	64^3	102^3	162^3
SP	12^3	36^3	64^3	102^3	162^3
BT	12^3	24^3	64^3	102^3	162^3

[10:12][61]

To overcome the first three of these problems, NPB 2 provided Fortran 77 source code using MPI for interprocess communication. Instead of pencil-and-paper specifications, this code was written to be very portable and to be representative of what a typical computational scientist might produce. The last shortfall was corrected by specifying another, larger problem class to supplement the originals [10:5-6,12]. Further, in 1997, a "workstation" problem class was specified for systems with less than 32 MB of memory [99].

Because the NBP 1 results still hold significance as what a system could achieve, NAS continues to accept NPB 1 results. For NPB 2 results, NAS defined three tiers [10:10]:

a) Unmodified – the only changes to the source code are those necessary to make the code execute.

b) Minor modifications – up to 5% of the lines of code are modified; modified source code must be provided.

c) Greater than 5% modifications – treated as NPB 1 results, except that non-vendor submissions are included with the NPB 2 results, and that modified source code must be provided.

The version of the NAS Parallel Benchmarks used in this thesis effort is NPB 2.3, downloaded from [99].

2.3.1. LU Simulated CFD Application

One of the simulated CFD applications in NPB 2 is LU, named after the format of the system of PDEs, and not because it uses LU decomposition (it doesn't [75:5]). Instead, the LU benchmark uses a well-known point-Gauss-Seidell relaxation scheme, SSOR,[18] to solve the three-dimensional compressible Navier-Stokes equations (Table 2-6) [78:13][101:1] using double-precision floating point arithmetic [61].[19] LU was selected over other applications as our testbed, as described in Section 3.2, because of three major factors:

a) it is designed specifically to have communication and computation patterns similar to "real" CFD applications [75:2];

[18] The SSOR algorithm is described in [11]. Given a system of PDEs expressed as $A\vec{x} = \vec{b}$, where A is the coefficient matrix, $\vec{b}$ is the vector of constants, and $\vec{x}$ is the solution vector, SSOR solves a system of PDEs by partitioning the coefficient matrix into upper & lower triangular matrices, then iterating through the formation of the constant vector, solving the upper triangle, solving the lower triangle, and updating the solution by calculating the steady-state residual [11:2]. The interprocess communication for current version of the parallel implementation is described in [101].

[19] Double-precision is the highest level of precision explicitly defined by the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, providing 15-17 base-10 digits of precision. The near-universal adoption of IEEE 754 assures that LU will provide identical results regardless of the platform on which it is executed [27:68-70].

b) it provides self-verification to establish that the solution is correct [10:7]; and

c) it is a well-known and easily-accessible piece of software, which makes it easier for others to reproduce our results or to compare their own results with ours.

Table 2-6. Navier-Stokes Equations.

$\frac{\partial \rho}{\partial t} + \nabla \cdot \left(\rho \vec{V}\right) = 0$	(2-1)
$\rho\left(\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_i}\right) = -\frac{\partial p}{\partial x_i} + B_i + \frac{\partial}{\partial x_j}\left[\mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial u_k}{\partial x_k}\right)\right] + \frac{\partial}{\partial x_j}\left(\zeta \frac{\partial u_k}{\partial x_k}\right)$	(2-2)
$\rho c_v \frac{DT}{Dt} = -p\nabla \cdot \vec{V} + \kappa \nabla^2 T - \nabla \cdot \vec{q}_r + \Phi + q'''$	(2-3)

[42:63]

In the unmodified LU code from NPB 2.3, the problem is partitioned among processors by alternately halving each processor's subdomain along the *x* and *y* axes (the *z* axis is not partitioned), resulting in a block checkerboard partitioning;[20] this requires a power-of-two number of processors [78:13-14]. Because Fortran 77 cannot allocate memory dynamically, this partitioning must be prepared at compile-time by specifying the problem class and the number of processors, allowing the correct amount of memory to be allocated [10:19]. Normally, this is completely appropriate – when using an MPP, or even a cluster with homogeneous nodes, static allocation of memory for equal-sized partitions provides the correct amount of memory needed by each process.

[20] See Section A.1.

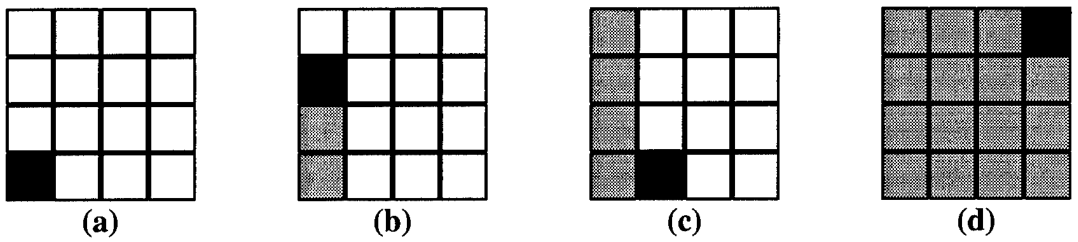


Figure 2-1. Relaxation of a tile. Active element is black; relaxed elements are gray; unrelaxed elements are clear.

The system of equations is solved by first defining each plane along the z axis on a particular subproblem as a tile. A tile is relaxed by starting in the corner grid point closest to the Cartesian origin; for the sake of discussion, the coordinates of this grid point are (i_0,j_0,k) (Figure 2-1a). Next, the (i_0+1,j_0,k) point is relaxed. Then, (i_0+2,j_0,k) is relaxed (Figure 2-1b). And so on, until the end of the column, $(i_{\max},j_0,k)$, is reached. This is repeated for the j_0+1 column (Figure 2-1c), the j_0+2 column, and each succeeding column until the last column of the tile, $j_{\max}$ is relaxed (Figure 2-1d) [101:5]. As a typical Gauss-Seidel relaxation process, the relaxation here uses first-order accuracy,[21] in which the value for (i,j,k) is found by making use of the six nearest neighbors, $(i\pm1,j,k)$, $(i,j\pm1,k)$, and $(i,j,k\pm1)$. Relaxation of (i,j,k) is only permitted after $(i\text{-}1,j,k)$, $(i,j\text{-}1,k)$, and $(i,j,k\text{-}1)$ have been relaxed, while the values from the previous relaxation of $(i+1,j,k)$, $(i,j+1,k)$, and $(i,j,k+1)$ are used [101:2].

Relaxation of the tiles begins with the tile closest to the origin, which we shall call (I_0,J_0,k_0) (Figure 2-2b). When tile (I,J,k) has been fully relaxed, the values[22] of its border

[21] See Section A.2 for discussion on solving continuous partial differential equations using discrete techniques.
[22] Density, energy, and momentum in the x, y, & z directions.

cells are communicated to the processors with tiles (*I*±1,*J*,*k*) and (*I*,*J*±1,*k*).[23] That complete, tiles (*I*+1,*J*,*k*), (*I*,*J*+1,*k*), and (*I*,*J*,*k*+1) are relaxed (Figure 2-2c). This process continues until all tiles have been relaxed (Figure 2-2*l*) [101:5]. This is one iteration; LU is known to converge to a solution in 250 iterations [75:5].

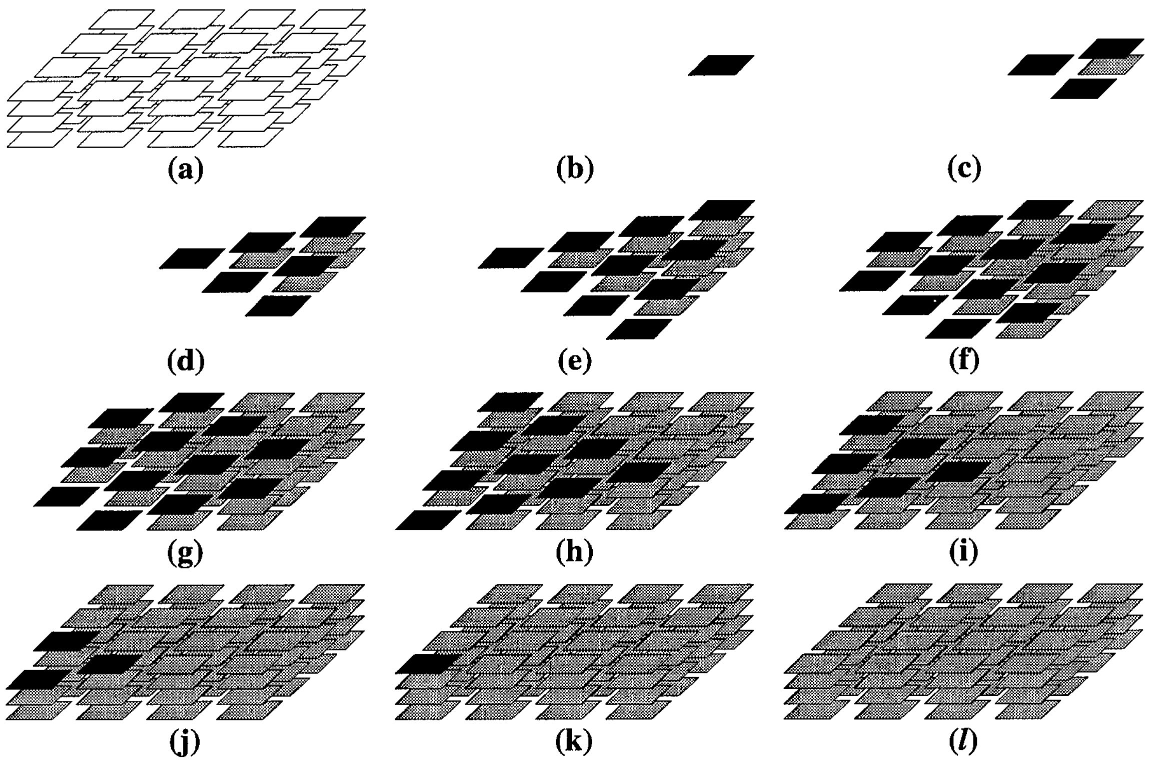


Figure 2-2. "Wavefront" of tile relaxation. Active tiles are black; relaxed tiles are gray; unrelaxed tiles are clear.

2.4. Load Balancing

Load imbalance is one of the major sources of overhead in a parallel system. Generally, this is because the nature of the application makes it extremely difficult, if not impossible, to predict the size of the subtasks *a priori*. Compounding this problem, the processors often must synchronize during execution; if all processors are not ready to synchronize at the same time, then those which are ready earlier must sit idle [48:135].

[23] Tiles (*I*,*J*,*k*±1) are on the same processor as (*I*,*J*,*k*).

Load balancing is the problem of minimizing the total idle processor time, and in so doing, minimizing the execution time of the complete parallel application. For a data-decomposed regular problem, such as CFD, load balancing typically is the near-trivial process of statically dividing the problem domain into equally-sized subdomains. This can even be done at compile-time, as the unmodified LU does [10:19]. For a task-decomposed problem or an irregular problem, dynamic load balancing must be used and is one of the most important modules in the application [22], as described in Section 2.4.2.

2.4.1. Load Balancing – Concept

Before going into further detail about load balancing, let us consider the concept and why it's important. Consider the contrived math problem:

> *Alone, Airman Jones can load a certain quantity of cargo onto an aircraft in twenty minutes. Airman Smith can load the same cargo in twelve minutes. Airman Banks can load the cargo in thirty minutes. How long would it take them to load the cargo together?*

This is essentially a parallel application – each of the airmen (processors) can work mostly independently of the others, with some time spent coordinating their actions. Ignoring the granularity of the problem of loading cargo, this is a straight-forward problem that a middle-schooler should be able to solve. If the airmen (processors) had the capability to shift work between them as needed, the cargo can be loaded in six minutes.[24]

But if they cannot adjust their work on the fly, or if it is too expensive (in terms of overhead), then they have to attempt to balance their workload statically. If their supervisor assumed they each are equally skilled cargo loaders, then they would each be assigned a third of the cargo. And if they are equally skilled loaders, this would be a good

[24] $\left({}^{1}\!/_{20\,\mathrm{min}} + {}^{1}\!/_{12\,\mathrm{min}} + {}^{1}\!/_{30\,\mathrm{min}}\right)^{-1} = \left({}^{10}\!/_{60\,\mathrm{min}}\right)^{-1} = 6\,\mathrm{min}$

decision, particularly because it is very inexpensive decision, computationally-speaking. However, since they aren't equally skilled loaders, the job would be finished in ten minutes, the amount of time it would take Banks to load a third of the cargo. Meanwhile, Jones has been relaxing for three minutes, twenty seconds, and Smith's been sitting around idle for six minutes. If their supervisor knew *a priori* the performance of each airman, then he could assign each an appropriate portion of the cargo and shave four minutes off the job.

2.4.2. Dynamic Load Balancing

In the above analogy, when the airmen are shifting their workloads without having to be told by their supervisor what each was responsible for, they are using dynamic load balancing. With dynamic load balancing, work is migrated from one processor to another to prevent processors from sitting idle while others are overworked. This can be achieved either by receiver-initiated techniques, in which idle processors request more work (Airman Smith finishes his portion of the cargo and offers to help Jones and Banks), or by sender-initiated techniques, in which processors with a load above some threshold seek processors with lesser loads to accept some of its load (Airman Banks asks Smith and Jones for help) [48:311,340].

The schemes to determine the donors and recipients, as well as the quantity of work to be migrated, are full areas of research in their own right and are beyond the scope of this thesis. The interested reader will find some are discussed in [48:313-315,317-321,340-341][28].

When the interconnection network (ICN) is a commodity network, dynamic load balancing becomes even more challenging due to the increased communication overhead involved. Kumar, *et.al.*, [48:320-321] mathematically treat this problem for a depth-first-search application, while Dubrovsky, *et.al.*, [28] at the Israel Institute of Technology treat it experimentally for four different applications.[25] Kumar, *et.al.*, treat only "simple" techniques like round-robin and random-polling, and do not contrast them with more "complex" techniques. On the other hand, Dubrovsky, *et.al.*, contrast round-robin with other strategies that try to find optimal task allocations and finds that the communication overhead of the "intelligent" strategies produce a greater overall runtime than the suboptimal allocations provided by round-robin; they found that the communication overhead of the "intelligent" techniques was too great for a commodity network, and so the "simple" strategies yielded better performance.

2.4.3. Asymmetric Static Load Balancing

In the cargo-loading analogy, when the supervisor assigned each airman (processor) a specific portion of the task, the supervisor is using static load balancing. While suitable for regular applications when each worker has equal capabilities, it should be clear from the analogy that if the assumption of equal capabilities is a erroneous, then the job requires more time to complete than is required. When the supervisor assigns each airman (processor) a portion of the task *according to his abilities*, he was using asymmetric load balancing.

[25] Matrix multiplication, all-pairs shortest path using Dijkstra's algorithm, solving a set of partial differential equations, and the Traveling Salesman Problem.

While static and dynamic load balancing for homogenous parallel computing platforms has been well studied for more than a decade, load balancing for heterogeneous parallel systems is a relatively new subject of investigation with less treatment [22]. On a heterogeneous platform, the goal is the same: to minimize idle processor time and, by extension, to lower the wall-clock time. This is done by distributing the work such that no processor is waiting for the completion of another [82]. The critical problem is that the load balancing techniques developed for homogenous systems are based on fixed parameters, tuned for the particular system. In a heterogeneous system, these parameters are not always known *a priori* [22].

Addressing this problem, researchers at the University of Paderborn [22] describe a *dynamic* load balancing technique that uses observed computational & communication performance to predict the time a task would complete on a given node and the time needed to query a node. Based on this technique, the authors developed new initiation, information exchange, and load exchange strategies that are suitable for a heterogeneous system [22].

Whereas the dynamic load balancing technique in [22] adjusts the load on nodes based on runtime performance, researchers at Brigham Young University [82] describe a *static* load balancing method that does not assign tasks until the abilities of the target nodes are known. This challenge is compounded by a variable system configuration – compute nodes are workstations "donated" to the system by logging onto a web page. The solution selected is to execute the HINT benchmark [39] once on each node to measure their capabilities and then to allocate the appropriate subtask. Each nodes' HINT results are stored for future use [82].

Finally, researchers at the Universidade de Coimbra in Portugal [81] ignored the problem. Like [82], compute nodes are donated workstations achieved by logging onto a web page. Unlike [82], the internet itself was the ICN, and donated workstations could be anywhere in the world. They recognized that dynamic load balancing was clearly out of the question due to the communication limitations. They also recognized that with such a dynamic system configuration, it is impractical to insist on knowing the capabilities of the donated workstations. Instead, their master process decomposes the problem into small and independent tasks (not necessarily the same size), which are farmed out to the worker processes on the workstations. When a worker process finishes its computation, it sends the results back to the master process and waits for its next simple task. Load balancing is achieved by reducing the problem to the finest granularity possible and never expecting a worker process to execute more than one simple task at a time [81]. The authors of [81] do not address the performance impact of this fine-grained task decomposition, as the ability to call upon the computing power of thousands of workstations should be viewed as an "enabling technology" rather than as a way to obtain performance speedup.

This survey, of course, is not the complete sum of all research in asymmetric load balancing, but it is representative, and it should convey to the readership the present level of research in this young field.

2.4.4. *Load Balancing LU*

Now consider Figure 2-3 and Figure 2-4, generated using the `upshot` profiling tool that is part of the MPICH distribution [9]. Instrumentation was added to the `ssor.f` file in the LU application to indicate when each process was computing an upper-triangle solution, a lower-triangle solution, or a steady-state residual. What we are observing are the rates at which two processors are executing portions of the SSOR code.

Figure 2-3 shows the time specific processors are spending in each portion of the SSOR engine of the LU application before asymmetric load balancing is implemented. Process 0 (P0) and Process 1 (P1) are identical processes, except for the data for which each is responsible. P0 is executing on a 450 MHz Pentium II, and P1 is executing on a 200 MHz Pentium; however, since traditional (symmetric) static load balancing is used, each has been assigned exactly half of the data set to solve.

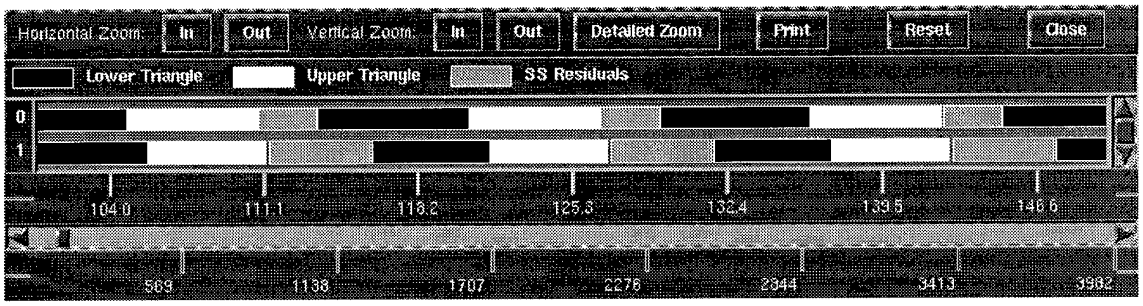


Figure 2-3. Unbalanced LU.A for 450 MHz Pentium II & 200 MHz Pentium.

Process 0 completes the Residual calculation faster than Process 1 does, and then it enters the portion of the code that deals with the lower triangle of the system of PDEs. P0 is immediately blocked for communication with P1 and must wait for P1 to reach the appropriate portion of the code. After the processors exchange information, they perform the lower triangle calculation; again, P0 completes the calculation faster than P1 and must wait for another information exchange before proceeding into the upper triangle region. This is why the faster node spends more time in the lower triangle segment than the slower one does – P0 finishes all calculations earlier but must periodically wait for P1 to catch up. The situation is similar for the upper triangle region.

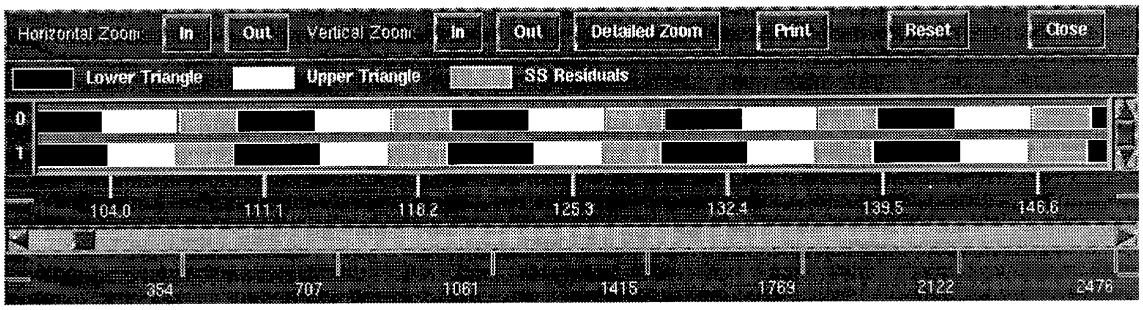


Figure 2-4. Load balanced LU.A for 450 MHz Pentium II & 200 MHz Pentium.

In Figure 2-4, a load balancing algorithm[26] is used to apportion the subdomains according to the processors' abilities. Whereas each process had previously been responsible for half of the domain, now P0 is assigned just over three-fourths of the problem, and P1 just less than a fourth. Clearly, the processes are now spending about the same amount of time in each section of the application and little time waiting for communication. As a result, when load balancing is used for this case, the application requires 48% less time to execute than the unbalanced version.

Compete results are presented in Chapter IV.

2.5. Summary

In this chapter, we reviewed the economic and technical aspects of the computer industry that led to the use of COTS hardware and software for high performance

[26] The load balancing algorithm developed in Section 3.3 using the Mflops weighting, described in Section 3.3.2.2.

computing. The current configuration of the ABC was described; details on its construction are in Section 3.1. Next, the application used to test the load balancing techniques developed in this thesis effort was discussed, followed by an explanation of load balancing and a review of previous efforts in asymmetric load balancing.

Now that we've seen a specific example of how load balancing can improve the LU application on a heterogeneous platform, we discuss how to implement and test load balancing in Chapter III.

III. Methodology

Building upon the background provided in Chapter II and Appendix A, we now consider the high- & low-level design and implementation decisions made in the course of this endeavor. After explaining why the LU application was selected as the testbed, we discuss the changes to the LU application, then the software created to measure the nodes' performance, and finally the design of the experiments.

3.1. Construction of the AFIT Bimodal Cluster

The AFIT Bimodal Cluster was started in the spring of 1998. There was no formal methodology to its construction, but rather a series of decisions that were addressed as they arose. One of the first considerations was whether the operating system should be Windows NT or Linux. Linux has the advantage that its license is free, and that the Beowulf project and related projects [52] had already broken the ground for parallel and distributed computing with Linux. Windows NT has the advantage that the nodes can be ordered with NT pre-installed, and that two members of our team are Microsoft Certified System Engineers (MCSE). We realized, though, that this need not be an either-or consideration, and both operating systems were included. The author undertook the responsibility of administering the Linux system, learning the job of system administration along the way.

Another early decision was that the most current hardware would be added to the ABC as money became available, and that personal computers already at AFIT that were offered to be part of the ABC would be considered on a case-by-case basis. The alternative was to use only homogeneous nodes, but this was ruled out for three reasons, two of which deal with future expansion. First, eventually, the processors used in the system would not be available for further expansion. Second, the ABC's performance would not be able to grow with technology. The third reason is that older computers donated to the project are additions to the ABC's capabilities without impacting our budget.

The initial Linux installation was not an advanced attempt at installation, because the author was still learning system administration. Four 333 MHz Pentium IIs were used, interconnected with an eight-port Fast Ethernet hub. Each had a full Linux installation, and each was essentially an independent computer; even NFS was not implemented, which meant users had separate home directories on each machine. Early testing with some simple kernels[27] revealed that the hub produced an unacceptable number of network collisions for communications-intensive applications, and a 24-port Fast Ethernet switch was ordered.

Another aspect of the system revealed during the early testing is an error in the `tcp_ack()` function in the Linux kernel which delays the transmission of partial packets, decreasing the network throughput (Figure 3-1). Whether this impacts performance, and by how much, depends on the application's communication patterns, particularly the size of the messages and how often they are transmitted. We have since learned of a fix [76], but chose not to implement the fix for temporal reasons.

[27] Matrix-vector multiplication, matrix-matrix multiplication, 1D FFT, and various sorting algorithms.

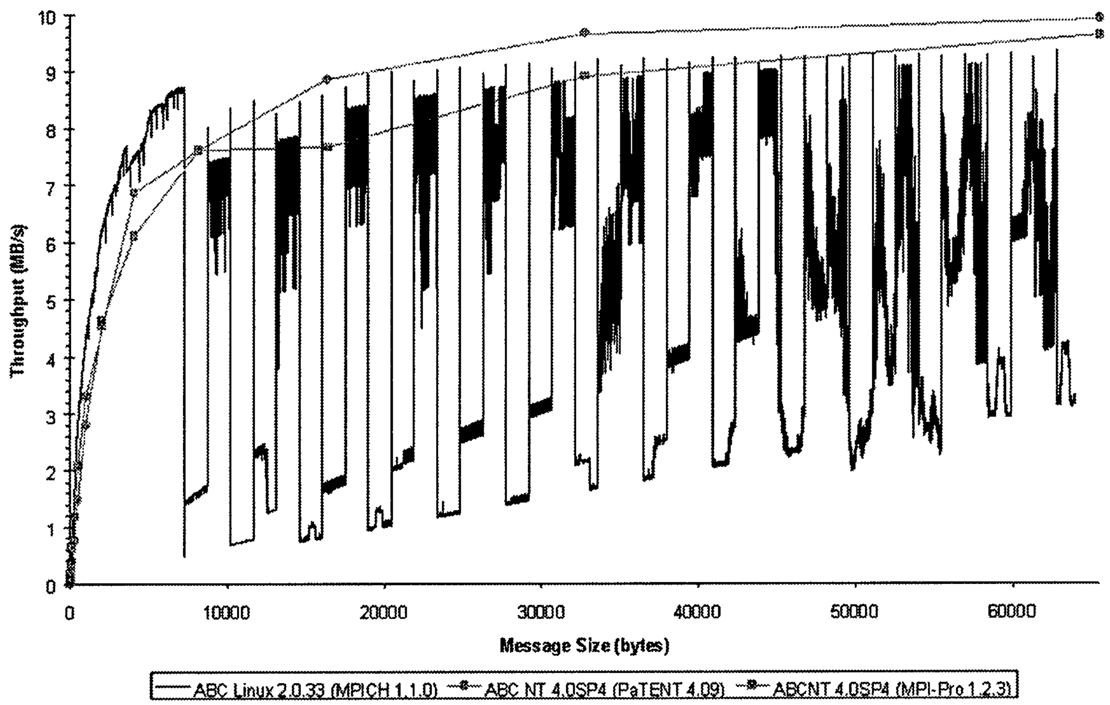


Figure 3-1. Delayed transmission of partial TCP packets under Linux.

When six 400 MHz Pentium IIs were added to the system, we took the opportunity to perform a full reinstallation of Linux, making use of lessons learned over the previous months. NFS was implemented to provide transparency to the users. We also sought to take advantage of the aggregate disk space within the cluster, should users need that much disk space, and NFS is used for this as well. We also attempted to implement NIS to share the password files across the cluster but were unable to get it to work correctly, and we decided to spend our time on issues more directly related to our research.

We had to address the issue of terminals for the system. When there were only four nodes in the cluster, each node had its own monitor, keyboard, and mouse. With ten nodes, physical space prevented this. Many other PoPC sites do not provide direct access to each node [66], but students using the Windows NT installation did require occasional access to each node. We solved this problem by ordering keyboard-video-mouse switches to allow the nodes to share terminals.

Security of the system is a serious issue. We address it by not letting it be an issue for Linux. The author, recognizing his limitations as a system administrator, has not configured the Linux installation to accept *any* remote access from outside the cluster. The cluster can be accessed remotely through the NT installation, but not through the Linux installation.

Tool selection is treated on an as-needed basis. For the early testing of the ABC, the gcc compiler was sufficient. However, the team determined that we also need a C++ compiler and a Fortran compiler. So, we downloaded the egcs suite [19], which includes C, C++, and Fortran 77 compilers, and we also ordered a commercial Fortran 90 license.[28] MPICH [9] was selected as the communications library because it was the one with which we already had experience.

Since these decisions were made, the cluster has grown further to include the 24-port switch, a 450 MHz Pentium II and a 200 MHz Pentium.[29]

3.2. Application Selection

We choose the computational fluid dynamics problem domain for this research because it is a well-understood deterministic problem, and because it would directly support other research at AFIT. With the problem domain selected, a specific application must be chosen on which to test the load balancing techniques. The three options

[28] The Fortran 90 license arrived too late to be used for this thesis.
[29] See Figure 1-1 and Table 2-2.

considered are: to write a simple CFD application, to use the NAS Parallel Benchmarks, and to use a "real-world" application.

Using a real-world CFD application would provide extra validity to our claim that asymmetric load balancing is suitable for real-world applications. We obtained such an application from the AFRL, but there are two reasons we do not use it. The first is that since it is not publicly available, other researchers would not be able to use the same application to reproduce our results. The second, more significant, reason is that the application is written using Fortran 90, and our Fortran 90 license is unavailable.

Writing a simple CFD application would not be hampered by either of the problems with using the AFRL code, but this option isn't the best option, either. First, it would require time to be spent developing and testing the CFD code instead of developing and testing the load balancing code. Second, the correctness of the algorithm would have to be assumed since the author, not being an expert on fluid dynamics, cannot verify the solutions obtained.

The NAS Parallel Benchmarks, though address all the above concerns. It is a publicly-available benchmark suite, most of which is written in Fortran 77. It also includes self-verification, which assures us that not only is the algorithm correct, but also that any modifications we make do not impact the correctness of the solution. It also reports both the time and the Mflops rate, which facilitates comparing the load balanced performance to the original code's performance [10:7].

Selection of the particular simulated CFD application from NPB is based on our desire to obtain as many points of comparison as possible. The SP and BT applications require a square number of processors [10:9]; given the processors available, this limits us to three quantities: one, four, and nine processors. LU, on the other hand, requires a power-of-two number of processors [10:8], permitting comparisons with four quantities: one, two, four, and eight processors. Further, at the time this decision was made, we still were using the 8-port hub, which would have prevented testing the nine-processor case for SP or BT. These factors made LU the preferable application.

3.3. LU Modifications

The full code of the LU application is not reproduced in this thesis due to its size[30] and because it is available for download from [99]. Appendix B, though, contains the output `diff` produces when contrasting the original and the modified code so other researchers can make use of these techniques and to facilitate the reproducibility of our results.

The modification paths are shown in Figure 3-2. Modification 0[31] was a minor modification to get the code to execute on the ABC, due to problems with MPICH 1.1.0 and Fortran, and does not represent a redesign of LU. The remaining changes are described in the following sections.

[30] There are 6265 lines of code in the original source files used by LU [61].
[31] See Section B.1.

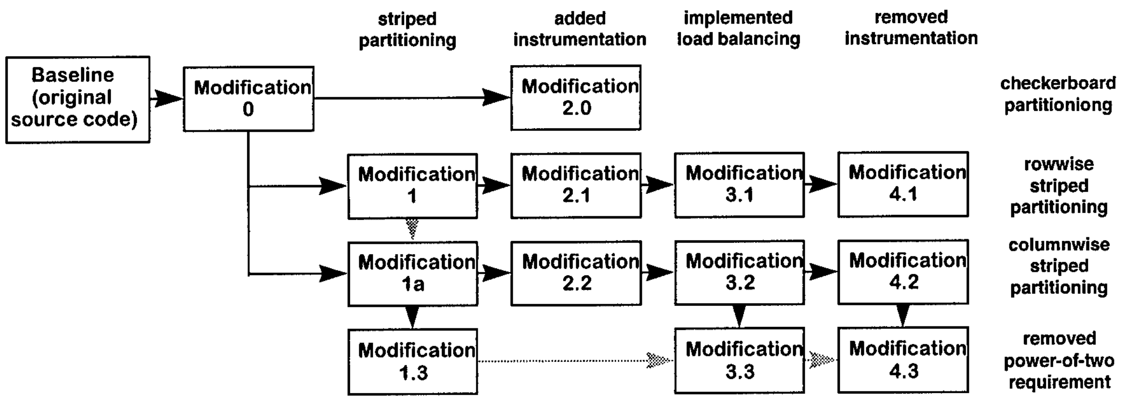


Figure 3-2. Modification progression of LU, from original to asymmetrically load balanced implementation.

3.3.1. Design of Domain Decomposition

3.3.1.1. Original Design

We begin by examining the relevant portions of the original LU code. We wish to study the domain decomposition independent of the implementation language, to avoid confusion caused by programming language constructs, such as `goto`'s. Z [44] lends itself well to this analysis by allowing us to clearly express the mathematical nature of the algorithm in set-and-logic notation. Programming-language-neutral models can also be created using Universal Modeling Language (UML) [74] and UNITY [15]. We choose not to use UML because the its object-oriented nature is not well-suited to the application we are describing. We opt for Z instead of UNITY for two reasons. UNITY does not specify on which processor an assignment takes place [15:9], yet to express the data partitioning, we must be able to express at least the proportion of data to be allocated to each processor. Also, Z has become one of the more popular specification languages [32:448], which increases the likelihood it will be familiar to the readership.

We first convert LU's original partitioning scheme[32] into Z syntax (Figure 3-3). The Z design begins with some simple constraints, namely that each process has a unique identification, the number of processes must be a power-of-two, each process can have at most one neighboring process in each of the cardinal directions, and each process can be a neighbor to no more than one process in each cardinal direction.

Process

$$
\begin{array}{l}
id : \mathcal{N} \\
north, south, west, east : \wp\ Process \\
nx, ny, nz : \mathcal{Z}^+ \\
ipt, jpt, kpt : \mathcal{N} \\
row, col : \mathcal{Z}^+ \\
isiz_1, isiz_2, isiz_3 : \mathcal{N} \\
\hline
\#north \leq 1 \\
\#south \leq 1 \\
\#west \leq 1 \\
\#east \leq 1
\end{array}
$$

(a)

Subdomain

$$
\begin{array}{l}
LU : \wp\ Process \\
nx_0, ny_0, nz_0 : \mathcal{Z}^+ \\
xdim, ydim : \mathcal{N} \\
\hline
\forall p,q \in LU, p \neq q \bullet \langle p.id \neq q.id \rangle \\
\forall p \in LU \bullet \langle p.id < \# LU \rangle \\
\forall p,q,r \in LU, p \neq q \bullet \langle (\#r > 0) \wedge (r = p.north) \Rightarrow (r \neq q.north) \rangle \\
\forall p,q,r \in LU, p \neq q \bullet \langle (\#r > 0) \wedge (r = p.south) \Rightarrow (r \neq q.south) \rangle \\
\forall p,q,r \in LU, p \neq q \bullet \langle (\#r > 0) \wedge (r = p.west) \Rightarrow (r \neq q.west) \rangle \\
\forall p,q,r \in LU, p \neq q \bullet \langle (\#r > 0) \wedge (r = p.east) \Rightarrow (r \neq q.east) \rangle \\
\forall p \in LU \bullet \langle p.row = \mathrm{mod}(p.id, xdim) + 1 \rangle \\
\forall p \in LU \bullet \left\langle p.col = \left\lfloor {}^{p.id}\!/_{x.\mathrm{dim}} \right\rfloor + 1 \right\rangle \\
\forall p \in LU \bullet \langle (4 \leq p.nx \leq p.isiz_1) \wedge (4 \leq p.ny \leq p.isiz_2) \wedge (4 \leq p.nz \leq p.isiz_3) \rangle
\end{array}
$$

(b)

Figure 3-3. Z design for original LU partitioning/load balancing.

[32] Mostly found in the file `subdomain.f` [61].

Unbalanced_Subdomain

Subdomain

$$\exists n \in \mathcal{N} \bullet \langle \# LU = 2^n \rangle$$

$$xdim = 2^{\left\lfloor \frac{\log_2(\# LU)}{2} \right\rfloor}$$

$$ydim = \frac{\# LU}{xdim}$$

$$\forall p \in LU \bullet \left\langle \begin{array}{l} \big(p.row \le \mathrm{mod}(nx_0, xdim)\big) \Rightarrow \\ \left(p.nx = \frac{nx_0}{xdim} + 1\right) \wedge \big(p.ipt = (p.row - 1) \cdot p.nx\big) \end{array} \right\rangle$$

$$\forall p \in LU \bullet \left\langle \begin{array}{l} \big(p.col \le \mathrm{mod}(ny_0, ydim)\big) \Rightarrow \\ \left(p.ny = \frac{ny_0}{ydim} + 1\right) \wedge \big(p.jpt = (p.col - 1) \cdot p.ny\big) \end{array} \right\rangle \qquad \textbf{(c)}$$

$$\forall p \in LU \bullet \left\langle \begin{array}{l} \big(p.row > \mathrm{mod}(nx_0, xdim)\big) \Rightarrow \\ \left(p.nx = \frac{nx_0}{xdim}\right) \wedge \big(p.ipt = (p.row - 1) \cdot p.nx + \mathrm{mod}(nx_0, xdim)\big) \end{array} \right\rangle$$

$$\forall p \in LU \bullet \left\langle \begin{array}{l} \big(p.col > \mathrm{mod}(ny_0, ydim)\big) \Rightarrow \\ \left(p.ny = \frac{ny_0}{ydim}\right) \wedge \big(p.jpt = (p.col - 1) \cdot p.ny + \mathrm{mod}(ny_0, ydim)\big) \end{array} \right\rangle$$

$$\forall p \in LU \bullet \langle p.nz = nz_0 \rangle$$

Figure 3-3 continued.

Beyond these constraints, the Z design is a series of predicates that define the block checkerboard row and column each process is responsible for, as well as the particular coordinates for the process' tiles

Every predicate in which there is only one instance of a process can be satisfied on each processor independent of the other processors. Nine of the twenty-one constraints are such predicates, offering parallelism in this symmetric load balancing scheme.

3.3.1.2. Design Changes

Having examined the baseline design, we consider the design considerations to implement asymmetric load balancing. The first consideration is the partitioning scheme. Section A.1.3.3 describes the alternatives.

Bearing in mind that this is an initial effort at asymmetrically load balancing LU, drastic changes such as irregularly-shaped tiles can be immediately ruled out. The next two options described, in which the row widths are varied within each column rather than globally, can also be ruled out; again, they require modifications to underlying assumptions in the CFD engine that, while suitable for future experimentation, are inappropriate for this early exploration into asymmetric load balancing.

This leaves two options left: block striping with variable column widths, and block checkerboard partitioning with the column and row widths balanced globally. Block striping is the simpler approach to implement, but that is not a sufficient reason to rule out the block checkerboard approach. The small number of processors available for this endeavor means that block checkerboard partitioning's advantage of using of all processors available is not a concern here.[33]

There are, however, two performance-related reasons to use block striping and not block checkerboard. First, based on the analysis in Section A.1.3.2, we expect that for smaller numbers of processors, the communication overhead involved in block checkerboard partitions is greater than that for block striped partitioning. Second, looking forward to implementation issues, the memory allocation requirements[34] drastically reduce the data locality for block checkerboard partitioning, which in turn increase cache misses and reduce the performance realized. Had this assessment proven erroneous, then block checkerboard partitioning could still have been implemented. As reflected in the next chapter, though, this is a good design decision.

[33] See Section A.1.3.1
[34] See Section 3.3.2.

The next issue considered is whether the block striped partitioning should be rowwise or columnwise. At an abstract level, such a decision could be arbitrary. For the LU application (assuming all dimensions are of equal magnitude), the communication requirements are the same, regardless of whether rowwise or columnwise block stripping is used. Likewise, the processor utilization would be no different, assuming data locality can be assured. And therein lies the key to this decision. Due to an *implementation* issue,[35] data locality would be severely hampered if rowwise block striping were used, and the performance penalty would be even greater than if block checkerboard partitioning were used. For this reason, columnwise block striped partitioning is the method of choice.

Figure 3-4 and Figure 3-5 provide the Z specification for the final partitioning scheme, implemented in modification 4.3. A "balanced process" is a process with the extra "weight" attribute; the nt_1 and nt_2 attributes are placeholders used to simplify expression of the specification without creating contradictions when specifying subdomain sizes.

Whereas in the baseline partitioning specification, nearly half of the constraints are parallelizable, not even a fifth of the constraints in the new partitioning scheme are. The implication is that there is not as much parallelism in the new partitioning scheme as there was in the original; many of the statements require global knowledge that was implicit in the original but now must be determined at runtime. This is not of great concern, as the time spent partitioning the domain is negligible when compared to the time involved in solving the system of PDEs.

[35] We use Fortran 77 to implement this software, which does not provide for dynamic memory allocation, which means more memory must be allocated for each partition than is needed. Since Fortran 77 stores multidimensional arrays in column-major order, rowwise striping loses data locality. See Section 3.3.2

Balanced_Process

Process

$nt_1, nt_2 : \mathcal{N}$ **(a)**

$weight : \Re$

$weight \geq 0.0$

Balanced_Subdomain

Subdomain

$S_1, S_3 : \wp\, process$

$t, u : process$

$\# LU \geq 1$

$\forall p \in LU \bullet \text{Is\_Balanced\_Process}(p)$

$xdim = 1$

$ydim = \# LU$

$\forall p \in LU \bullet \langle p.weight = \text{WeighNode}(p) \rangle$

$\forall p \in LU \bullet \langle p.nx = nx_0 \rangle$

$$\forall p \in LU \bullet \left\langle p.nt_1 = ny_0 \cdot \text{round}\left(p.weight \Big/ \sum_{q \in LU} q.weight \right) \right\rangle$$ **(b)**

$\forall p \in LU \bullet \langle p.nz = nz_0 \rangle$

$$\left(\sum_{p \in LU} p.nt_1 = ny_0 \right) \Rightarrow \left(\forall p \in LU \bullet \langle p.nt_2 = p.nt_1 \rangle \right)$$

$$\left(\sum_{p \in LU} p.nt_1 > ny_0 \right) \Rightarrow \left(\begin{array}{l} \left[S_1 \mathrel{\hat=} \{ p \mid (p \in LU) \wedge (\forall q \in LU \bullet \langle p.weight \leq q.weight \rangle) \} \right] \wedge \\ [t \in S_1] \wedge \text{CorrectDown}(t, \varnothing) \end{array} \right)$$

$$\left(\sum_{p \in LU} p.nt_1 < ny_0 \right) \Rightarrow \left(\begin{array}{l} \left[S_1 \mathrel{\hat=} \{ p \mid (p \in LU) \wedge (\forall q \in LU \bullet \langle p.weight \geq q.weight \rangle) \} \right] \wedge \\ [t \in S_1] \wedge \text{CorrectUp}(t, \varnothing) \end{array} \right)$$

$$\left(\forall p \in LU \bullet \langle p.nt_2 \geq 4 \rangle \right) \Rightarrow \left(\forall q \in LU \bullet \langle q.ny = q.nt_2 \rangle \right)$$

$$\left(\exists p \in LU \bullet \langle p.nt_2 < 4 \rangle \right) \Rightarrow \left(\begin{array}{l} \left[S_3 \mathrel{\hat=} \{ p \mid (p \in LU) \wedge (\forall q \in LU \bullet \langle p.weight \leq q.weight \rangle) \} \right] \wedge \\ [u \in S_3] \wedge \text{Debalance}(u, \varnothing, 4 - u.nt_2) \end{array} \right)$$

$\text{AssignPosition}(0, LU)$

Figure 3-4. Z design for final LU partitioning/load balancing.

A tile's width is determined by multiplying the size of the global domain's *y* dimension, ny_0, by the processor's fraction of the total computing power. Because fractional elements are nonsensical, this product is rounded to the nearest natural number. If, however, there are rounding errors, then one or more subdomains must have their size corrected. If the sum of the subdomain *y* dimensions is greater than the global domain's *y* dimension, then the correction is achieved by reducing the load of the weaker processors. Conversely, the most powerful processors are given a greater load if the rounding errors resulted in too few elements.

Next, if any tile widths are less than four elements, then some of the load balancing is undone. The specification for a minimum width is a legacy requirement from the original software [61], presumably to keep the communication overhead from dominating the application, and it was retained for precisely that same reason. Whereas the original LU had no way to prevent a tile from being fewer than four elements wide, the load balanced version has the recourse of correcting a violation of this constraint that had been caused by the load balancing process itself.

Finally, after all corrections have been made and *ny* is known for each process, the precise boundaries of each tile can be established.

CorrectDown

ΞLU
$a?, a! : process$
$b? : \wp\, process$
$S_2 : \wp\, process$
$c : process$

$S_2 \triangleq \{p \mid [p \in (LU \setminus (\{a\} \cup b))] \wedge [\forall q \in (LU \setminus (\{a\} \cup b)) \bullet \langle p.weight \le q.weight \rangle]\}$ **(a)**
$c \in S_2$
$\left(\sum_{p \in LU} p.nt_1 = ny_0 - \#b\right) \Rightarrow (a!.nt_2 = a?.nt_1)$
$\left(\sum_{p \in LU} p.nt_1 > ny_0 - \#b\right) \Rightarrow (a!.nt_2 = a?.nt_1 - 1)$
$\text{CorrectDown}(c, \{a\} \cup b)$

Figure 3-5. Supporting functions for Figure 3-4.

CorrectUp

ΞLU
$a?, a! : process$
$b? : \wp\, process$
$S_2 : \wp\, process$
$c : process$

$S_2 \triangleq \{p \mid [p \in (LU \setminus (\{a\} \cup b))] \wedge [\forall q \in (LU \setminus (\{a\} \cup b)) \bullet \langle p.weight \geq q.weight \rangle]\}$
$c \in S_2$
$\left(\sum_{p \in LU} p.nt_1 = ny_0 + \#b \right) \Rightarrow (a!.nt_2 = a?.nt_1)$
$\left(\sum_{p \in LU} p.nt_1 < ny_0 + \#b \right) \Rightarrow (a!.nt_2 = a?.nt_1 - 1)$
$\text{CorrectUp}(c, \{a\} \cup b)$

(b)

Debalance

ΞLU
$a?, a! : process$
$b? : \wp\, process$
$diff\ ? : \mathcal{N}$
$b? : \wp\, processS_2 : \wp\, process$
$c : process$
$left : \mathcal{N}$

$S_2 \triangleq \{p \mid [p \in (LU \setminus (\{a\} \cup b))] \wedge [\forall q \in (LU \setminus (\{a\} \cup b)) \bullet \langle p.nt_2 \leq q.nt_2 \rangle]\}$
$c \in S_2$
$left = c.nt_2 - diff\ ?$
$$(\#(LU \setminus b) > 1) \Rightarrow \begin{pmatrix} [(left \geq 4) \Rightarrow (a.ny = a.nt_2 + diff) \wedge (c.ny = left)] \wedge \\ [(left < 4) \Rightarrow ((a.ny = a.nt_2 + diff) \wedge \text{Debalance}(c, \{a\} \cup b, 4 - left))] \end{pmatrix}$$
$(\#(LU \setminus b) = 1) \Rightarrow (a.ny = left)$

(c)

AssignPosition

$tpt? : \mathcal{N}$
$S_4?, S_4! : \wp\, process$
$d : process$
$S_5? : \wp\, process$

$S_5 \triangleq \{p \mid [p \in S_4?] \wedge [\forall q \in S_4? \bullet \langle p.ny \leq q.ny \rangle]\}$
$d \in S_5$
$d.jpt = tpt?$
$(\#S_5 > 1) \Rightarrow \text{AssignPosition}(d.jpt + d.ny, S_4? \setminus \{d\})$

(d)

Figure 3-5 continued.

3.3.2. Low-Level Design & Implementation

3.3.2.1. Low-Level Design of Partitioning and Load Balancing

The implementation language selected, we turn to implementing block striped partitioning. As detailed in Section A.1.2, implementing the partition poorly can have a serious performance penalty if data locality is lost. Fortran stores multidimensional arrays in column-major order [1:10]. This means that if we allocate the full domain size but the columns are not the length of the full domain, then the process' memory access patterns are brief periods of unit stride followed by leaps across the memory space.

To put numbers to this, if the full domain has $64 \times 64 \times 64$ elements, but a process' row is only four elements wide, and each word in the arrays is eight bytes, then the process strides through 32 bytes and then skip over 480 bytes before it reaches the next element of the array that it can use. *At best*, this causes a reduction in cache hits, and the application suffers a performance penalty. Consider, though, that the A-class problem size for LU requires 40 MB[36] just for the problem domain, and that ABC12 has only 32 MB of main memory. As the process executing on ABC12 relaxes a tile,[37] it cannot avoid thrashing to swap space.

On the other hand, if the columns are as long as the full domain, then unit stride memory access is possible, thereby making greater use of data locality. And, if ABC12 is responsible for half (or less) of the problem domain, then the columns for which it is not responsible remain in swap space, never to be touched. Clearly, since Fortran 77 is the

36

$$64^3\,elements \times \frac{20DP\,FP\,words}{element} \times \frac{8bytes}{DP\,FP\,word} = 5\times 2^{3\times 6+2+3}bytes = 5\times 2^{23}bytes = 40\,MB$$

[61]

37 See Section 2.3.1.

implementation language for this investigation, then columnwise block striped partitioning is a must. However, to quantify the performance loss incurred due to poor partitioning, both columnwise and rowwise block striped partitioning were implemented.

Referring again to Figure 3-2, we examine the incremental changes made to LU in the course of this investigation. Each modification is a minimal change. This is to facilitate the isolation of errors by requiring testing of only small portions at a time.

The change from modification 0 to modification 1 is a change in the partitioning scheme from block checkerboard to block striped. Specifically, modification 1 introduces rowwise block striping. Even if we did not wish to quantify the penalty associated with poor partitioning, the progression from LU's original block checkerboard partitioning to rowwise block striping is a safe first-step. The original LU's partitioning was such that there are no fewer rows than columns [61]. Changing to rowwise block striping is as straight-forward as changing the equations that define the dimensionality of the tiles. Modification 1a[38] is also an implementation of block striped partitioning, this time columnwise. Besides changing the equations that define the tiles' dimensions, other portions of code also needed to be modified to remove the assumption that the rows are no wider than the columns. LU's self-verification is useful in establishing that all necessary changes have been made, and that the changes have not affected the correctness of the algorithm.

The 2.x modifications add instrumentation to the code, providing us with insight to the partitioning and how much time each process spends in certain portions of the code.

[38] See Section B.2.

The 3.x modifications implement the load balancing algorithms, and the 4.x modifications deinstrument the code.

Examination of modification 4.2[39] shows the implementation of the design specified in Figure 3-4 and Figure 3-5. Each process weighs its node in parallel with the others, and then an all-to-all exchange is made to give each process global knowledge about the system's capabilities. Once this piece of knowledge is available to each process, two implementation options are available: we could either make maximal use of the available parallelism, or each process could make the partitioning calculations for each process and use only what it needs. Because each process eventually needs to know the precise location and size of the partition on its lower-numbered neighbor (a requirement that recurses down to Process 1), then if each process calculates only its own partitioning we need to engage in more interprocess communication. Instead, we make note that these calculations are inexpensive and that our ICN is not a high performance network. For these reasons, we require each process to conduct the full calculation of the partition assignments.

Most of the calculation can be mapped straight from the Z specification, using iterative loops instead of recursive functions. One particular requirement, though, requires some cleverness. We must be able to establish an ordering of the processes by their weight. Obviously, a sort is required. The problem, though, is that we must also preserve the original ordering of the processes as well. Simply copying the process' information and then sorting the copies is insufficient, as we want changes in the process' attributes to be reflected in the original. Some indirection is required, and the solution should be familiar to any C programmer: pointers. Dereferencing the pointers allows us to sort the pointers based on the weights of the processes to which they point.

The last modifications (1.3, 3.3, and 4.3[40]) remove the requirement that the number of processors be a power-of-two. The original requirement was necessary to enable block-checkerboard partitioning, but with block striped partitioning, this is no longer required. After this last change, any number of processors up to the maximum permitted can be utilized.

3.3.2.2. Language Selection

Once the low-level design has been established, a critical implementation decision is the implementation language itself. The LU application was written in Fortran 77 with a few common extensions. The NPB 2 programmers had considered Fortran 90 but ruled it out due to performance concerns associated with Fortran 90 [10:6][27:85-85,285-286]. Unfortunately, Fortran 77 does not support dynamic memory allocation, unlike Fortran 90. This means that should Fortran 77 remain the implementation language, sufficient memory must be provided to each process to accommodate the largest subdomain it might be assigned [27:282].

The alternative to allocating enough memory on each processor for the entire problem is to use a different implementation language. Obviously, the prime criteria for an alternate implementation language are that it support dynamic memory allocation and that it be able to either link with the MPI libraries [9] or link with a "wrapper" routine written in a language that can link with the MPI libraries. Rewriting the program in C (or C++) would be an inefficient use of development time that could be used better elsewhere.

[39] See Section B.3.

Other non-Fortran languages have the same disadvantage, and possibly others. The remaining options, then, are Fortran 77 with oversized arrays and Fortran 90. Since a Fortran 90 license is unavailable for the experiments, Fortran 77 is the implementation language by default.

The other complication with implementing the low-level design using Fortran 77 instead of Fortran 90 or C/C++, besides its inability to allocate memory dynamically, is the lack of pointers, which we specify as a necessity in the previous subsection. We overcome the problem by creating pseudo-pointers. These are not real pointers, but rather an array that hashes to the array with process information. Instead of sorting real pointers, we sort the elements of the hashing array based on the weights of the processes to which they map. That done, we continue to use the hashing array to adjust the partition assignment for the least and/or most powerful processors.

3.4. Measurement of Compute Node Performance

Any asymmetric load balancing algorithm must have some way to determine the performance capabilities of each node. Decker, *et.al.*, [22] use the run-time performance of the application to adjust the load dynamically. Silva [81] gets around the issue by decomposing the application into the finest grain possible. For our application, the approach in [22] is not suitable because it uses dynamic load balancing, and we are using static load balancing which, by definition, precludes knowledge about the run-time performance. The solution [81] uses is unacceptable due to the performance penalty such a fine-grained decomposition would have.

[40] See Section B.4.

```
      isiz0t = isiz01
      iarg1 = 1                                ! /proc/cpuinfo converted
      iarg2 = 1                                ! /proc/cpuinfo converted
      iarg3 = 4                                ! /proc/cpuinfo converted
      weight = weighnode(iarg1,iarg2,iarg3)    ! /proc/cpuinfo converted
      call MPI_ALLGATHER(weight,1,MPI_DOUBLE_PRECISION,
         glblw8,1,MPI_DOUBLE_PRECISION,
         MPI_COMM_WORLD, IERROR)
      ttlw8 = 0.0
      do 3651 loop=0,nnodes_compiled-1
         ttlw8 = ttlw8 + glblw8(loop)
 3651 continue
      sum = 0
      do 3652 loop=0,nnodes_compiled-1
         temp = glblw8(loop)*isiz0t             ! common subexpression
         nt(loop) = temp/ttlw8                  ! nt is int, so truncated
         if (mod(temp,ttlw8)/ttlw8.ge.0.5) then
            nt(loop) = nt(loop)+1               ! correct rounding error
         endif
         sum = sum+nt(loop)                     ! to check the math later
         pointer(loop) = loop                   ! initialize pointers
 3652 continue
      sorted = .false.
 3655 if (.not.sorted) then
         sorted = .true.
         do 3656 loop=0,nnodes_compiled-2
            if (nt(pointer(loop)).gt.nt(pointer(loop+1))) then
               itemp = pointer(loop)
               pointer(loop) = pointer(loop+1)
               pointer(loop+1) = itemp
               sorted = .false.
            endif
 3656    continue
         go to 3655
      endif
      lo_end = 0                                ! steal from the poor
      hi_end = nnodes_compiled-1                ! give to the rich
      if (sum.ne.isiz0t) then                   ! nuts
 3657    if (sum.gt.isiz0t) then                ! ease the lowend's load
            nt(pointer(lo_end)) = nt(pointer(lo_end))-1
            lo_end = lo_end+1                   ! share the easement
            sum = sum-1
            go to 3657                          ! make sure we're finished
         endif
 3658    if (sum.lt.isiz0t) then                ! more work for highend
            nt(pointer(hi_end)) = nt(pointer(hi_end))+1
            hi_end = hi_end-1                   ! share the extra effort
            sum = sum+1
            go to 3658                          ! make sure we're done
         endif
      endif
      do 3659 loop=0,nnodes_compiled-2
         if (nt(pointer(loop)).lt.4) then       ! nuts
            itemp = 4-nt(pointer(loop))
            nt(pointer(loop)) = nt(pointer(loop))+itemp
            nt(pointer(loop+1)) = nt(pointer(loop+1))-itemp
         endif
 3659 continue
      if (nt(pointer(nnodes_compiled-1)).lt.4) then ! gosh darn it
      endif                    ! do nothing ... it'll get caught below
      tpt(0) = 0
      do 3654 loop=1,nnodes_compiled-1
         tpt(loop) = tpt(loop-1)+nt(loop-1)
 3654 continue
      ny = nt(id)
      jpt = tpt(id)
```

Figure 3-6. Asymmetric load balancing implementation.

Instead, we consider the technique used by Snell, *et.al.*, [82] who rely on a preliminary run of HINT on each node to achieve this, with the option of storing each nodes' result on a local file to avoid recalculation of the QUIPS rating in the future.[41] The problem with this approach is that it is computationally expensive. Even with storing the result for future use, it neither scales well, nor is it cheaply portable. If new nodes are introduced to the system, they cannot be used for processing until they have been benchmarked with HINT first. Considering that the system in [82] is a network of workstations that are dynamically donated and removed by the workstations' owners, this introduces a great deal of overhead when a workstation is first donated. Further, if the application is ported to a new system, then the entire system must now be rated with HINT.

So the question, then, is can the relative capabilities of the nodes be estimated without the expense of running HINT (or some other comprehensive benchmark) on each node first? If so, how? And, how effective is it? To answer these questions, we must reconsider the nature of the underlying hardware and operating system.

3.4.1. Design

3.4.1.1. Amortizing the Computational Cost of Classifying Nodes

The first observation we make is recognizing that while the system as a whole is heterogeneous, many nodes are similar, even identical, to each other. If we could take advantage of this knowledge, then we have already reduced the overhead of using a benchmark such as HINT.

[41] See Section 2.4.3.

The first approach we consider is to identify the unique types of nodes and run the benchmark on one of each type. Then, recognizing that every node has a unique identification, we could build a map from node id to benchmark rating. In the case of the ABC, where there are four types of nodes, twelve nodes total,[42] this would cut the overhead of running the benchmark down to one-third of the processor time. When we add new nodes to the system, we can add them to the map if they are identical to a node already in the cluster. If they are not identical to a pre-existing node, then we can execute the benchmark on them and then add them to the map, which is still no worse than the method in [82].

While we have reduced the overhead induced by adding nodes to the system, there are still problems which must be addressed. The most serious problem is that this approach requires human editing of a file – building the map cannot be done by the system unless it already knows which nodes are identical to each other, and to know which nodes are identical to each other, either a human must provide that information, or the system must determine that information by measuring the capabilities of each node. Which brings back the original dilemma, our desire to avoid running an expensive benchmark for each individual node. Related to this problem, is that this approach is unsuitable for dynamic addition of nodes. If a node is donated for the first time without the map being prepared first, then the system must still measure the new node's performance. Finally, we haven't improved the ease of porting to a new system! Unless the maps are published with documentation explaining that a node with a certain processor clocked at a certain rate has

[42] See Figure 1-1 and Table 2-2.

a certain rating, then researchers using a different system must build their own map from the ground-up.

With Linux, though, we can actually create a map that associates system information with benchmark ratings. Linux, like all UNIX systems, has a directory in its directory tree called `/proc`. The files in `/proc` do not reside on disk, but rather are created by the OS and reside in main memory [98:22]. One of these files, `cpuinfo`, contains information such as the processor's manufacturer and model. While it does not include the rate at which the processor is clocked, it does include a performance-related value called "BogoMIPS." BogoMIPS, meaning "bogus MIPS," is calculated when Linux boots to calibrate certain timing loops used elsewhere [92]. It has been described as "how many times the computer can do nothing in one second" [33:19]. What benefits us is that when combined with the manufacturer and model information, BogoMIPS allows us to identify each unique type of processor. This means the map can be generated by the computer and that new nodes that are identical to existing nodes can be mapped without human intervention. Further, a map can be placed on another system without editing and still be useful. Finally, if a new node is added that is not mapped, then the system can use the information known about other nodes to estimate the mapping for the new node without executing the benchmark on that node. The primary shortfall is that `/proc/cpuinfo` is not available on other operating systems, not even other versions of UNIX.

3.4.1.2. Benchmark Selection

The next design consideration is the benchmark selection. We first want the benchmark to accurately predict the performance our application realizes on our nodes.

Second to this, we want the benchmark to be a general indicator of performance so that other programmers can make use of its measurements.

3.4.1.2.1.LINPACK

LINPACK [62], once *the* indicator of floating point performance had stressed the floating point and memory performance of computer systems by solving a dense system of linear equations. A variation continues to be used to gauge supercomputers for the Top500 list [89]. However, modern microprocessors are able to hold the entire data structure[43] in today's larger caches; this has largely made LINPACK yet another meaningless indicator of performance [27:332-334]. So LINPACK neither provides a good prediction of our nodes' performance, nor was it ever designed to gauge integer performance, thus limiting its usefulness as a general metric.

3.4.1.2.2.NPB

There are sequential versions of NPB 2 [99] available, and they would certainly be a good measure of the performance we can expect out of the LU application. While generality is a secondary consideration, using NPB 2-serial to adjust the load for an NPB application is *too* specific, and doing so could provide unrealistic expectations for real-world applications.

3.4.1.2.3.SPEC

The SPEC benchmark suite [83] is considered to be the best general indicator of performance for modern microprocessors. Both integer and floating point versions are available, which means the suite is general enough to be used for other applications. And

[43] 320 KB.

the most current floating point version, SPECfp95, includes benchmarks[44] that should reveal very good indications of the performance we can expect with our application [27:340-344]. The downside is that, in the interest of keeping the benchmarks from being overtaken by improved hardware and/or compiler customizations, the SPEC benchmark suite is updated every few years. Thinking beyond the immediate thesis effort, switching to the new suites when they are released would require rebuilding the map for the entire system, yet failing to upgrade to the new suites would result in poor measurements for the newer hardware. Nevertheless, this is the best option of those considered so far.

3.4.1.2.4.HINT

HINT, though, has advantages over SPEC. HINT is a memory-oriented benchmark that constrains neither problem size, number of iterations, or running time. It reports "quality improvements per second" (QUIPS) that is determined by calculating the area under a curve to finer and finer degrees of precision. In this fashion, the problem continues to grow until no further improvements in the calculation can be realized. The benchmark can be compiled for *any* intrinsic data type, including integers using 8 bits, 16 bits, 32 bits, and 64 bits, and floating point numbers using 32 bits, 64 bits, 80 bits (on Intel processors), and 128 bits (on processors that support quad-precision). The QUIPS rating reflects the processor performance for whichever data type is being evaluated, the memory hierarchy from L1 cache to swap space, unit- and non-unit-stride memory performance, and numerical accuracy. So, HINT can provide a "good" indication of a computer's performance for memory-bound computation-intensive applications,

[44] Mesh generation, shallow water simulation, partial differential equations, Monte-Carlo computation, fluid-dynamics, multigrid solver in three-dimensional potential field, turbulence modeling, weather prediction, quantum chemistry, and Maxwell's equations.

regardless of its dominant data type. In particular, HINT's results have been shown to correspond well with NPB's results [82]. We conclude, then, that HINT is a good metric both for our specific case and for general use as well.

HINT stores the results of its calculations to disk to permit plotting the performance as a function of memory usage or execution time, and it also reports a single QUIPS value that is the integral of that plot [27:339]. It is this single QUIPS value that we are interested in.

3.4.1.2.5.Quick 'n' Dirty Benchmarks

One of our desires is to be able to measure the nodes' capabilities with as little overhead as is possible. So far, we have discussed doing this by amortizing the cost of measuring the nodes' capabilities. The method we are using to achieve this amortization is by creating a map from system information to the benchmark results. This rather forces us to ask ourselves if we're not overlooking something obvious. One of the pieces of information available to us from `/proc/cpuinfo` is the OS' measure of how fast a certain kind of busy loop runs [92]. As long as we're already obtaining this information, it costs us nothing but processor time to determine if BogoMIPS can be used to provide effective and *cheap* load balancing. In a similar vein, we can create an inexpensive routine that loops through a series of floating point operations to create a crude Mflops rating. This has the additional benefit that it is not dependent on the OS. A fall-out of being usable on a non-Linux OS is that we can also use this crude Mflops rating to index the benchmark results on other systems.

3.4.2. Implementation

3.4.2.1. Language Selection

As with the application, we must consider the implementation language for the measurement library. With the application, the driving consideration was that the application was already written in Fortran 77. With the metric library, we do not modifying preexisting code; rather, the library is new software, free from the constraints of others' implementation decisions. The only consideration is functionality.

The first element of functionality is that it must function with the application. That is, the Fortran code must be able to call the metric code and receive useful data back. When Fortran calls a subroutine or function, it uses call-by-reference [29:96-98]. Obviously, writing the metric library in Fortran would satisfy the requirement that the application code be able to interface with the metric code. But what alternatives are there? We address Ada 95, Java, and ANSI C/C++.

We can immediately rule Ada 95 out. Some Ada compilers use call-by-copy/restore, while others use call-by-reference [86:70]. This ambiguity does not facilitate integrating the software with the Fortran code. Further, if call-by-copy/restore is used, then we cannot interface the application software with the metric software. Java passes intrinsics by value and objects by reference [91:44]. Since intrinsic data types are passed between the application and the metric software, we can now also rule out Java. Besides the obvious choice of Fortran, we are now left to consider C and C++. Both C and C++ are call-by-value, except that the value passed can be a memory address, which effectively provides for call-by-reference as well [86:69-70].

In the interest of abstracting the application programmer from the details of the metric software, we would like the metric library to offer a single interface to the application software that accepts certain parameters to define which measurement technique should be used. In this fashion, the application programmer needn't be concerned with the semantics of the functions that actually perform the measurements.

Since the combination of arguments to the front-end varies, depending on the metric technique preferred, we must have some way to deal with the different arrangements of parameters. C++ (and Java & Ada 95) provide for function overloading [20:73]. C's `stdarg` library includes variable argument functions which can result in convoluted code to properly interpret the arguments [46:462-463]. No mention of variable arguments with Fortran was found in our literature review. The problem with C++ (as well as Java & Ada 95) is that they are object-oriented, and Fortran is not. As such, Fortran can not interface with the name-mangled code produced by an object-oriented compiler.

Another issue is that we want to provide for identical syntax, regardless of the calling language. Fortran compilers append one or two underscores after subprogram names and assume that subprograms are compiled similarly. C compilers do not append underscores to subprogram names and assume the subprograms are compiled likewise. If the front-end (`weighnode()`) is compiled with Fortran, then an application programmer using C must be prepared to call `weighnode_()` or `weighnode__()`, while a Fortran application programmer calls `weighnode()`. This may be considered a minor irritation, and one that can easily be incorporated into programmers' mindset, but it violates our desire that application programmers be offered identical syntax such that they need not be concerned with the language in which the library is written. On the other hand, we need not provide such an abstraction to the library programmer. If we write the library in C, then we can create three front-end functions, `weighnode()`, `weighnode_()`, and `weighnode__()`, and specify that the application programmer always pass the arguments by reference. Then, whether the library be called from a C application or a Fortran application with either trailing underscore setting, and the application programmer always addresses it as `weighnode()`, not caring about the library's implementation language.

Finally, the implementation language must be able to actually perform the measurement. Both C and Fortran, can execute the simple floating point test proposed in Section 3.4.1.2.5, naturally. Similarly, both are able to parse an ASCII text file, which is how `/proc/cpuinfo` appears to the program. But we would like to be able to use HINT – or any other benchmark we might want to try in the future – without having to modify its source code. So our implementation language must be able to issue commands to the system to initiate the benchmark, and it must be able to create a pipe to read the system's `stdout`. C's intimacy with UNIX (and by extension, Linux) [46:1-2] makes it ideally suited to interfacing with the OS.

Clearly, C is the best option for implementing the NodeMetric library. It can interface with either a Fortran or a C application; it can provide for uniform semantics to the application programmer; it has facilities to interpret variable arguments; and it can interface with the OS to obtain the benchmark results without forcing us to recode the benchmark.

3.4.2.2. Implementation of NodeMetric Library

As mentioned in the last section, we wish the NodeMetric library to have a single interface to the application with common semantics, regardless of the application language. This front-end to the library is are the `weighnode()` functions, found in Section C.1. All the information the application programmer needs is available in the `weighnode.h`[45] file. The `weighnode()` functions interpret the arguments provided by the application programmer to call the appropriate functions that actually measure the nodes' performance.

The original intent was to write only one function that calls the functions which do the measurement, and the other front-end functions would receive the parameters from the application and pass them in-turn to the primary front-end. Doing so would have simplified code maintenance, in that bug-fixes and the addition of features would require the modification of only one function. The problem, though, is that the greatest coding effort is in decoding the variable arguments. The secondary front-ends would not be able to blindly forward the arguments to the primary front-end; they must first determine which arguments and of what type.[46] Adding features, and possibly fixing errors, would require changes to the variable argument decoding. So long as the secondary front-ends must fully decode the arguments, there is no value in them calling the primary front-end. At this point, code maintenance can be more easily achieved by making modifications to one of the functions, testing, and then copy-and-pasting the function body into the other two functions. This is precisely how the `weighnode()` functions were developed.

[45] See Section C.1.1.
[46] The use of void pointers was attempted to avoid this obstacle, but this attempt did not solve the problem.

Based on the arguments passed to it, the `weighnode()` functions takes one of nineteen possible actions:

a) For an invalid argument combination, they return a weight of zero to indicate an error. This value was chosen because it draws attention to the error even if the application programmer does not check for it, *e.g.*, through division-by-zero.

b) If the application programmer wishes, the front-end functions return a floating-point value that the application programmer passes to the front-end function. The utility of this feature is that it allows programmers to specify the load balancing, regardless of the weights the NodeMetric library could return, without altering the application structure. They might use this to give all nodes equal weights, for testing purposes, or they might find some other function that they believe provides a better metric and, again, they don't wish to change the application to test it.

c) The front-end functions can call an internal function to execute the HINT benchmark for one of five datatypes, a short integer (short), a long integer (int), a long long integer (longlong), a single-precision real (float), or a double-precision real (double). We don't expect this option to be exercised much.

d) The `weighnode()` functions can call the internal function that parses the `/proc/cpuinfo` file. Optionally, it passes the results to another internal function that maps the first function's output to the QUIPS value provided by one of the five datatypes for which we compiled HINT.

e) The functions may call the function that provides a very simple test of the processor's floating point performance. In turn, this result also can be passed to a function that maps to a QUIPS value for one of the datatypes.

The library's internal functions are located in two files, metric.c[47] and metricmap.c.[48] In addition to these files are header files metric.h and metricmap.h, which contain the function prototypes; the header files can be included in any file, such as weighnode.c, that call the internal functions. The header file nodeinfo.h provides the data structure for the mapping functions, as well as functions to read and save the maps. Finally, buildmap.c is the only file that is intended to produce an executable file; its purpose is to build and expand the files that map from the inexpensive benchmarks to the HINT results. Discussion of buildmap.c is unnecessary, as the same issues are addressed during the discussion about metricmap.c.

3.4.2.2.1.metric.c

Three functions are included in metric.c: parse_cpuinfo(), calc_pi(), and run_hint().

There is little left to discuss concerning parse_cpuinfo(). It opens /proc/cpuinfo for read-only, reads the file until the string "bogomips" is found, reads in the BogoMIPS value, closes the file, and returns the BogoMIPS value. This version does include some assumptions that do not hold for the general case.

First, parse_cpuinfo() assumes there is only one processor per node – the function only reads the first BogoMIPS value it finds. This shouldn't be a problem, since all processors within an SMP node should be identical. This version also assumes the manufacturer and model of the processor are irrelevant. Different implementations of the

[47] See Section C.2.
[48] See Section C.3.

IA may provide different BogoMIPS ratings, which can be misleading. For example, the branch prediction scheme in AMD implementations result in higher BogoMIPS values than an equivalent Intel processor [92]. Further, according to van Dorst [92], similarly-clocked Intel 80486 and Pentium processors have similar BogoMIPS, yet the Pentium's floating point unit is far, far superior to that of the 80486 [14:622,679]. At this stage in the ABC's development, this assumption is not a problem – the difference between the FPU in the P5 processor core in the Pentium and the P6 processor core in the Pentium II is not as dramatic as is the difference between the 80486 and the P5 [14:679,699]. Should new implementations of the IA be introduced to the ABC, then `parse_cpuinfo()` will need to be adjusted to consider the make and model and then scale the BogoMIPS value appropriately before returning.

The next function, `calc_pi()`, times a series of floating point calculations and reports back the number of floating point operations completed per second. Making use of the "constructive laziness" adage, that "it's almost always easier to start from a good partial solution than from nothing at all" [68:3], we find code that can be reused from the MPICH [9] distribution. Included in the example code with MPICH is a file that calculates the value of pi by determining the area under a curve. Removal of the parallel constructs from its kernel gives us a very simple test of the processor's FPU and branch prediction scheme (as well as the compiler's ability to optimize small loops).

This function requires between one and thirty seconds to execute, depending on the number of iterations of the loops are executed. In testing calc_pi, we find that 2^{22} iterations provides a good value in about a second – more iterations yield similar values but require more time, and fewer iterations provide results that are not as good.

As an interesting side-note, when `weighnode()` calls `calc_pi()`, an address for `calc_pi()` to place the result of the calculation must be passed as an argument. Without this argument, an optimizing compiler would (and did!) recognize that pi is not visible outside the function, so there is no need even to calculate pi, and the entire function is optimized down to timing the empty space between two calls to `clock()`.

The sole purpose of `run_hint()` is to launch the HINT benchmark and retrieve the single QUIPS rating that HINT provides. The initial effort is geared towards creating a child process that calls `execv()` to transmogrify into the HINT executable code. We are unwilling to change the source code for the benchmark because we would like to be able to implement other benchmarks in the future with minimal effort; this means we cannot explicitly pipe the output from the child to the parent process, the challenge is to capture the child process' `stdout`. It doesn't take long to discover that we are making our task more difficult than it need be.

We can make use of C's ability to issue commands directly to the operating system. In particular, we can make use of C's ability to create a pipe between a program and the system command [46:397-398]. By opening a read-only pipe to the operating system when we issue the command to launch HINT, the information HINT places in `stdout` can be parsed by `run_hint()` as though it were a file, allowing us to obtain the QUIPS value.

3.4.2.2.2.metricmap.c & buildmap.c

The functions `convert_parse_cpuinfo()` and `convert_calc_pi()` each map the result of a simple metric to the result of a previous execution of the HINT benchmark. Implementation considerations for these functions and the `buildmap` executable are so intimately related that we shall consider them both in this section.

The first consideration is the data structure for the map files – "smart data structures and dumb code works a lot better than the other way around" [68:7]. Our first effort is implementing an elegant almost-complete binary search tree (BST) that finds the appropriate mapping in $\mathcal{O}(\log_2 n)$ time and stores the BST on disk and in memory in such a fashion as to minimize disk access time and make good use of cache for large data sets. In the process of isolating bugs, we realize that for small n, there isn't an appreciable difference between searching in $\mathcal{O}(\log_2 n)$ time and $\mathcal{O}(n)$ time, and that for small data sets, the elegant solution would spend more time accessing the disk than would reading the entire data set at once. For this reason, we implement a simple linear list instead. This does not appear be a bad decision, as testing indicates mapping the metric functions' outputs to a QUIPS value adds only about a second to the process of weighing a node.

Each element of the list has seven fields – the key that the metricmap functions compares against their input, five values corresponding to the QUIPS that HINT produces for five datatypes, and a field to indicate the size of the list.

Recognizing that the input to the metricmap functions might not correspond exactly to one of the keys in the list, we decide how to deal with this eventuality. Ignoring the problem and returning an error is not an acceptable solution. Instead, we make the reasonable assumption that if the input is between two known values, then the output must likewise be between the mapped outputs of the two known values. Lacking any better knowledge about the exact relationship between the input values and the output values, we linearly interpolate to get the output. What about extrapolation? If the input is outside the range of known values, what action should be taken? Again, ignoring the problem is not viable. Inputs less than the smallest known value can easily be treated by interpolating between the smallest known value and zero – this decision is made implicit in the mapping functions by making the smallest value in the list zero. Inputs greater than the largest known value are not so easily dismissed. Extrapolating beyond the range of known values is not a safe practice if the nature of the relationship between inputs and outputs is not known. We certainly don't want to overcorrect the partitioning so as to make the load balance problem worse. For this reason, we do not extrapolate beyond the largest known value; inputs greater than that value are mapped to that largest known value's output.

Finally, because the `calc_pi()` function can produce different outputs on the same processor, as a function of the number of iterations it progresses through, and as a function of other random activities on that processor, it is unwise to map a single value produced by `calc_pi()` to each QUIPS value. In the same vein, it is unwieldy to map every value `calc_pi()` produces to each QUIPS value for that processor. Instead, we take advantage of the interpolation we already incorporated. When adding a processor to the map, we execute `parse_cpuinfo()` once (since it always returns the same value), but we execute `calc_pi()` eight times over the range of "good" iteration values that were determined empirically. The largest and smallest of these values are used as keys to the QUIPS values for that processor; any runtime values that fall between this minimum and maximum have an output "interpolated" between two identical QUIPS ratings.

3.5. Design of Experiments

The first question is the size of the problem size we wish to test. The "sample" and "workstation" classes can be dismissed as unsuitable for these tests because they do not sufficiently task the system. This leaves problem sizes A, B, and C. We have already demonstrated that the A-class problem only fits on ABC12 by placing some of the unused memory allocation in swap space.[49] The B-class problem requires 162 MB,[50] and ABC12 has only 96 MB virtual memory total.[51] Since the B-class cannot be used with ABC12, we therefore use the A-class to test our load balancing algorithm.

We could blindly execute the software on every combination of two, four, and eight processors possible. This is not only undesirable, it is unnecessary. Since the processors are not all unique, we need not use every possible combination of processors to fully characterize the system, we could just use every unique combination of 200 MHz, 333 MHz, 400 MHz, and 450 MHz processors. Given the finite time available for experiments, even this is undesirable. Instead, we need to consider exactly what we wish to learn. We desire to learn how the load balancing algorithm developed in Section 3.3 affects the performance of the application for different combinations of processors. We can achieve this by looking at different ranges of capabilities.

At one extreme, we would include both the 450 MHz processor and the 200 MHz processor. At the other extreme (for the two- and four-processor cases), we would use only the 450 MHz processor and 400 MHz processors. Between the two extremes, the least powerful node would be a 333 MHz processor. By looking at ranges of capabilities instead of combinations of processors, we reduce the number of combinations to test

[49] See Section 3.3.2.2.

down to eight: three two-processor combinations, three four-processor combinations, and two eight-processor combinations.

The next question is which versions should we test? Naturally, we execute the unmodified code to compare results against those produced by code whose performance we haven't affected. Since our load balanced code uses striped partitioning, we also execute the unbalanced code with striped partitioning. To be able to account for the performance impact of overallocating memory for the partitions, we execute the code with this overallocation, but with each processor reporting identical weights to the load balancing algorithm. Finally, we test the three weighting approaches for load balancing, BogoMIPS, Mflops, and QUIPS. To reduce the cost of using QUIPS, we use our library's ability to map from the BogoMIPS rating to the QUIPS rating.

A critical issue is how to measure performance, so that we can determine if, and how much, the performance has improved. Any computer engineer should emphasize that time is the one true measure of performance – the system that obtains the same (correct) solution in the least time is the fastest [64:52]. Using "million instructions per second" (MIPS) is generally unsuitable since it varies with the number of instructions used to obtain the solution, the instruction mix, and when comparing different platforms, the instruction set and clock rate [64:60-61]. "Million floating point operations per second" is only slightly better since it specifies the type of instructions we're interested in; however, it is still a function of the algorithm used and the underlying platform [64:64-65]. The authors of [34] make the case for "quality improvements per second" as a computer system metric, but it is particular to the HINT benchmark and cannot be used to measure

[50] $102^3/64^3=4.05$. The B-class requires 4.05 times the memory required by the A-class.

the performance of an application. We shall use Mflops as our metric to compare the performance with different partitioning schemes since the underlying platform is fixed, the algorithm is fixed, and the LU application reports the Mflops sustained during execution. Our load balancing algorithm does not affect the total number of floating point operations, so in this case, Mflops is proportional only to the inverse of time. The reason we choose not to use execution time is because there are results in which the execution time of one case is several thousand seconds while the execution time of each of the other cases in the test is a couple hundred seconds. When graphing these results, if time is used, then the very slow case forces a scaling that hides the relative performance of the other cases. If Mflops is used, then the very slow case is represented as a very small value and does not affect the scale of the graphs.

We also wish to know whether load balancing allows us to make use of the weakest processor in the cluster, or if the performance achieved with load balancing is still worse than the performance achieved without using that processor at all. This leads to tests using one, three, and seven processors in combinations that match the broadest combination of two, four, and eight processors, except for the absence of the 200 MHz processor. This is not possible with the unmodified code's checkerboard partitioning, but the versions with striped partitioning can still be used for these tests.

We wish to characterize the performance of the system with more than eight processors in use. For this reason, we execute the code on all twelve processors and on

[51] See Table 2-2.

eleven processors (excepting the 200 MHz processor). Feedback from the results of these tests[52] lead to tests with other numbers of processors greater than eight.

The final two questions deal with statistical validation of our results. We need to determine how many executions of each test should be performed. Ideally, we run the tests twenty or thirty times to obtain small confidence intervals.[53] However, because the time the tests require – particularly for the tests with fewer processors – we instead choose to execute each test five times, leaving the option open to run more tests if some results are statistically ambiguous.

After conducting the experiments, we must analyze and present the results. When presenting the results in graph form, we use box plots because they provide a visual impression of the location, spread, and skewness of data sets, and they are particularly useful for comparing multiple data sets [57:206].

When comparing the performance obtained with load balancing against that obtained without, we always test the set of values obtained with load balancing against the best performance obtained without. Likewise, when determining whether load balancing permits improved performance by adding a weak processor, we compare against the best performance obtained without the extra processor.

We do not report speedup using the traditional definition, "the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processors" [48:118], because we are not executing LU on identical processors. Instead, the speedup we report is the ratio of the

[52] See Section 4.5.
[53] The size of the confidence interval is inversely proportional to the square root of the number of data points [57:265].

time taken to solve a problem on *p* heterogeneous processors without asymmetric load balancing to the time required to solve the same problem on the same processors with asymmetric load balancing.

We must conduct statistical tests to conclude whether the load balanced performance is an improvement over the best unbalanced performance or not. The problem that arises is that with five data points, we cannot neglect the question of whether the performance results are normally distributed [57:277]. Examination of the performance box plots suggests the data is not normally distributed, but the Lilliefors test for normality [57:278-280] cannot establish that the data is not normally distributed, either. We must, therefore, err on the side of caution, and use a test that does not rely on a normal distribution; specifically, we use the Wilcoxon signed rank test [57:285-288].[54] For the cases where the load balanced performance for all five data points is always greater than (or always less than) the best unbalanced performance, the Wilcoxon signed rank test allows us to conclude that the load balancing has (or has not) improved performance with a 0.03125 level of significance.

Executing the tests hundreds of times until the results are tightly clustered around the "true" values and outliers can be isolated and dismissed would be ideal. Given that this is impractical, we use statistical tests to validate our results so other researchers can make use of the software and techniques developed in this thesis effort to make more efficient use of the AFIT Bimodal Cluster and other heterogeneous clusters of PCs.

[54] In the analysis, significance values are obtained from http://fonsg3.let.uva.nl/Service/Statistics/Signed_Rank_Test.html .

3.6. Summary

We opened this chapter with a discussion about how the AFIT Bimodal Cluster was developed. Next, the selection of the application was addressed, followed by the changes made to the application. This discussion began by considering the design of LU's original decomposition algorithm and progressed into the design and implementation of the partitioning algorithm we wished to test.

Next, we looked at measuring the performance of the compute nodes, beginning with a discussion on how to do this better than others have in the past, followed by a discussion on what our options are to provide the measurement, and ending with a look at implementation issues. Wrapping up the chapter, we presented an outline of the experiments that test our load balancing scheme.

IV. Results & Analysis

In the last chapter, we discussed the development of the load balancing software and the design of the experiments to test the software. This chapter presents the results and analysis of those experiments.

This chapter is organized as follows: Sections 4.1 through 4.4 presents the results of the major experiments conducted for this thesis. Section 4.5 addresses the performance of LU on the ABC beyond eight processors. Finally, Section 4.6 discusses some results tangential to the focus of this thesis effort, namely NaN exceptions that are generated for certain cases, the effect of non-unit stride memory access, and the experimental price-performance ratio.

4.1. One-Processor Results

Load balancing is nonsensical when the load can be assigned only to one processor. The original intent was to run the one-processor case five times for only the 450 MHz Pentium II node with checkerboard partitioning and both rowwise and columnwise striped partitioning, as a baseline for other comparisons. However, when NaN exceptions were found to have impacted the results of the four-processor and single-processor checkerboard partitioning cases,[55] the single-processor checkerboard and columnwise striped partitioning cases were rerun five times on one node of each type[56] to determine if a particular node was producing faulty results, perhaps as a result of bad memory.

[55] See Section 4.6.1.

Due to the extra demand of page faults, ABC12 had not yet completed the second execution of LU by the time the other processors had finished their single-processor runs, and the queued executions of LU on ABC12 were terminated. The results of this test are discussed in Section 4.6.1.

4.2. Two-Processor Results

Three combinations of two-processors are tested; these combinations are the 450 MHz Pentium II with either the 400 MHz Pentium II, the 333 MHz Pentium II, or the 200 MHz Pentium. The Checkerboard partitioning case is run only once for each combination, as

a) we were being cautious about spending processor time on tests that result in NaNs;

b) the other cases which had *not* generated NaN exceptions indicated that the striped partitioning was producing better performance than the checkerboard partitioning,[57] and our comparison for load balancing speedup uses the best unbalanced performance; and

c) for the two processor case, there is no difference in the partitioning between the checkerboard and rowwise striped partitioning,[58] and when the exact amount of memory is allocated for rowwise striped partitioning, there is no performance difference between it and columnwise striped partitioning.[59]

All other two-processor tests used the full five executions, as described in Section 3.5.

[56] ABC03, ABC09, ABC11, ABC12.
[57] See Section 4.4
[58] The LU application checkerboard partitioning divides the rows before it divides the columns [61].
[59] See Section A.1.2.

4.2.1. 1x200 1x450

Figure 4-1 is a box plot showing the megaflops performance results of the six tests conducted for the two-processor case, specifically for the case in which a 450 MHz Pentium II and a 200 MHz Pentium are used. The boxes show the interquartile ranges, and the "whiskers" extend to the limits of the data.[60]

The "Unbalanced Checkerboard" plot is for the unmodified data partitioning scheme, and the "Unbalanced Col Striped" plot shows the results of changing the partitioning to columnwise striped partitioning, without implementing asymmetric load balancing. "Equal Weight" is from code that has been modified to permit asymmetric load balancing, but each node is weighted the same; this permits us to assess the performance penalty induced by allocating more than sufficient memory on each node. The last three plots, "B'MIPS Weight," "Mflops Weight," and "QUIPS Weight" are the results of applying the three different weightings to the asymmetric load balancing algorithm.

[60] As with all the cases tested, all values for this case are within the bounds of the inner fences.

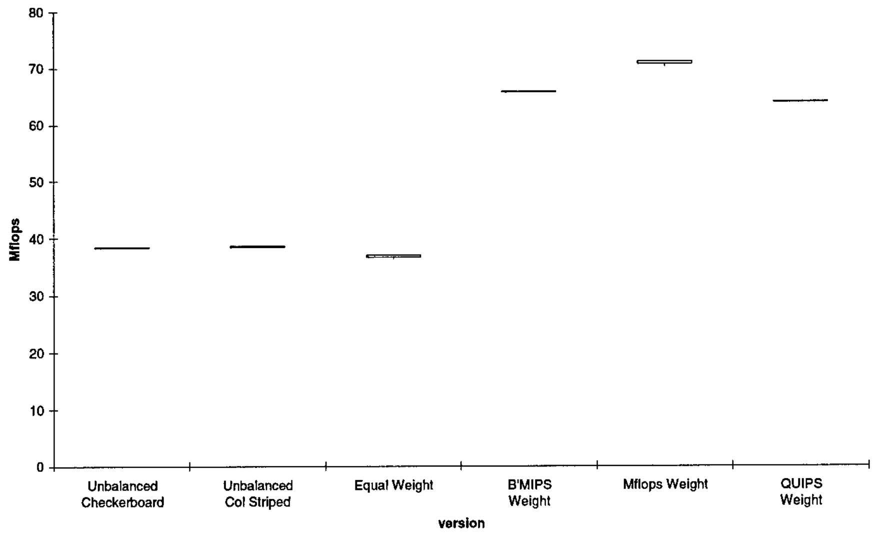
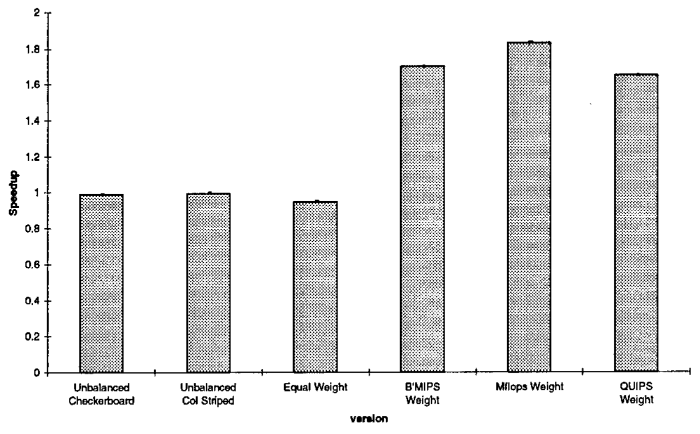


Figure 4-1. Performance with & without load balancing – 2 processors, 1x200 1x450.

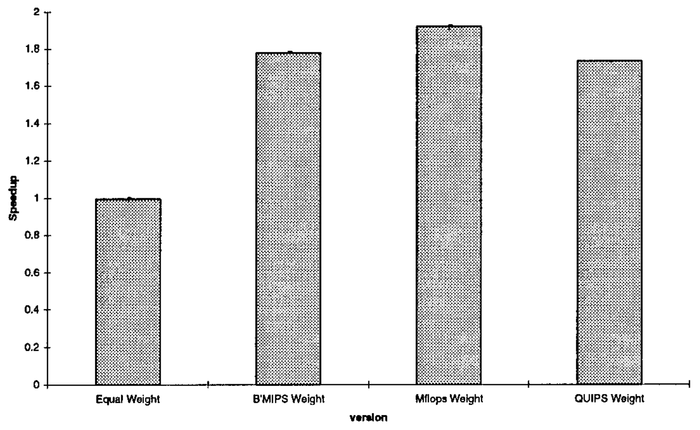
Casual inspection of the Figure 4-1 shows that with processors as disparate as a 200 MHz Pentium and a 450 MHz Pentium II, we realize a substantial performance improvement, and as explained in Section 3.5, the Wilcoxon signed rank test tells us that for each of the weightings, there is nearly a 97% probability that the true value of the load balanced performance is greater than the best unbalanced performance.

A contrast between the unbalanced case and the "equal weight" case reveals the penalty for excessive memory allocation is, on average, 5.0%, with a standard deviation of 0.72%. As can be seen in Figure 4-2a, the load balancing scheme provides a performance boost of between 64.9% and 83.4% over the best unbalanced performance. This breaks out as 69.8±0.17% improvement using the BogoMIPS weighting, 82.9±0.95% improvement using the Mflops weighting, and 65.0±0.03% improvement using the QUIPS weighting.

Accounting for the memory penalty, we observe in Figure 4-2b that the load balanced code performs 72.9% to 92.3% better than the "equal weight" code with the memory penalty. Specifically, 78.0±0.18% for BogoMIPS, 91.7±1.00% for Mflops, and 72.9±0.03% for QUIPS.



(a)



(b)

Figure 4-2. Speedup over best non-load balanced performance – 2 processors, 1x200 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

Given that there is such a difference in the capabilities of the two processors, might we have been better to not have used the Pentium? The answer, in this case, is clearly no.

The best performance on a single 450 MHz Pentium II is 60.17 Mflops. The worst performance on the 450 MHz Pentium II and 200 MHz Pentium, when load balancing is used, is 63.93 Mflops. Granted, this is an improvement of only 6.2%, but this is also contrasting the best uniprocessor performance with the worst load balanced two-processor performance. Using the best weighting (in this case the Mflops weighting), we realize a 17.8±0.61% improvement over the best uniprocessor performance.

4.2.2. 1x333 1x450

If instead of a 200 MHz Pentium, we use a 333 MHz Pentium II in conjunction with a 450 MHz Pentium II, we obtain the performance indicated in Figure 4-3. Two aspects are immediately obvious. First, the performance gain over the unbalanced code is not appreciable. Second, the memory penalty is greater.

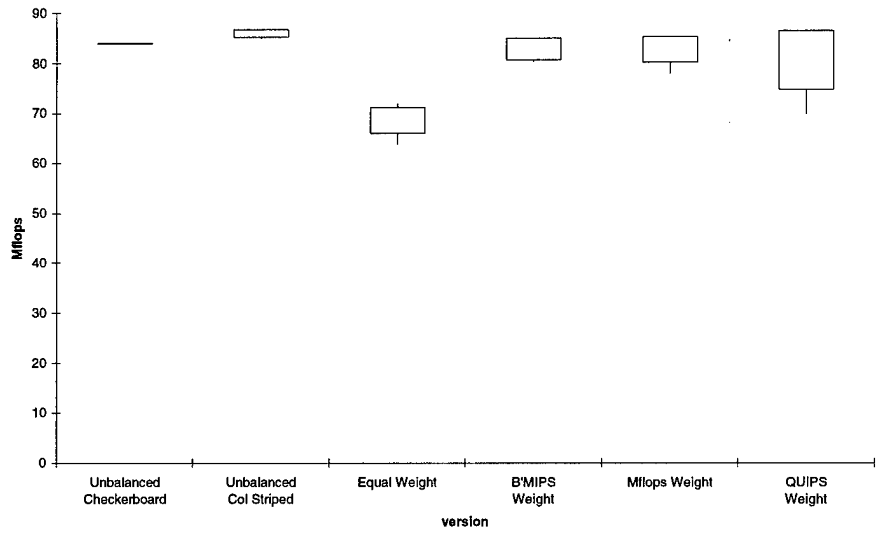


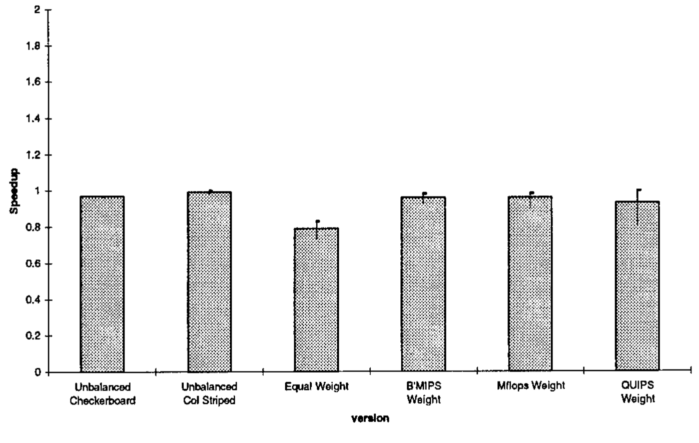
Figure 4-3. Performance with & without load balancing – 2 processors, 1x333 1x450.

Examining Figure 4-4, we can quantify those observations. Not only is the performance gain over the unbalanced code unappreciable, it is nonexistant. The best load balanced performance is still 0.4% shy of the best unbalanced performance.

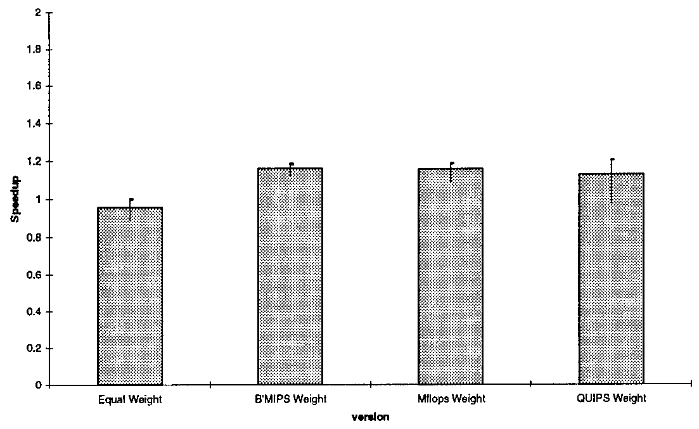
With this combination of processors, the penalty for excessive memory allocation is 20.9±3.55%. Using the best performance of the equally-weighted partitioning with memory penalty as a baseline, the BogoMIPS weighting provides an improvement of 15.8±3.31%, the Mflops weighting offers a 15.7±4.32% improvement, and the QUIPS weighting improvement is 12.5±9.55%.

4.2.3. 1x400 1x450

The last set of two-processor tests use two processors with similar capabilities, namely a 400 MHz Pentium II and a 450 MHz Pentium II (Figure 4-5).



(a)



(b)

Figure 4-4. Speedup over best non-load balanced performance – 2 processors, 1x333 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

As with the tests in Section 4.2.2, the load balanced codes underperform the unbalanced code by 12.5±9.55% collectively. This is not surprising, considering the memory penalty averages 18.1%. Taking the memory penalty into account, though, the load balancing algorithm provides performance improvements between of 4.3±0.13%, 4.5±0.12%, and 4.9±0.06% (Figure 4-6b).

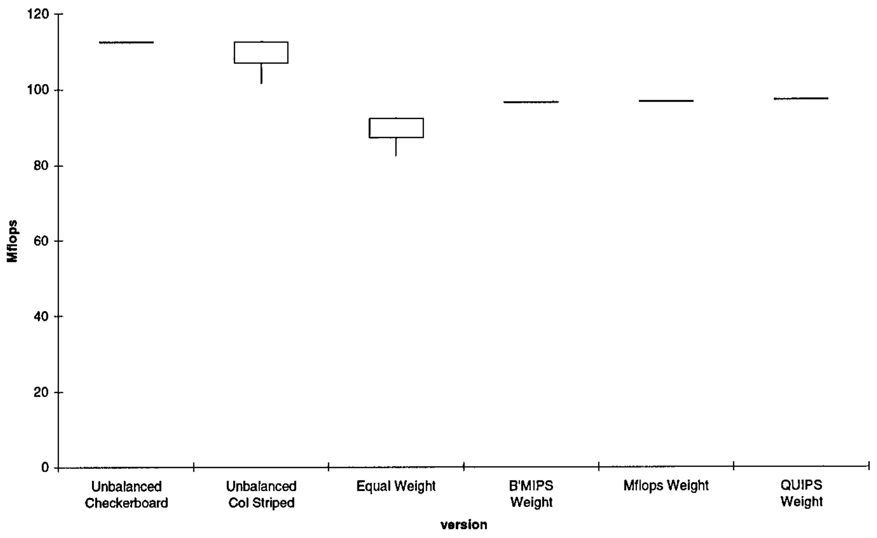
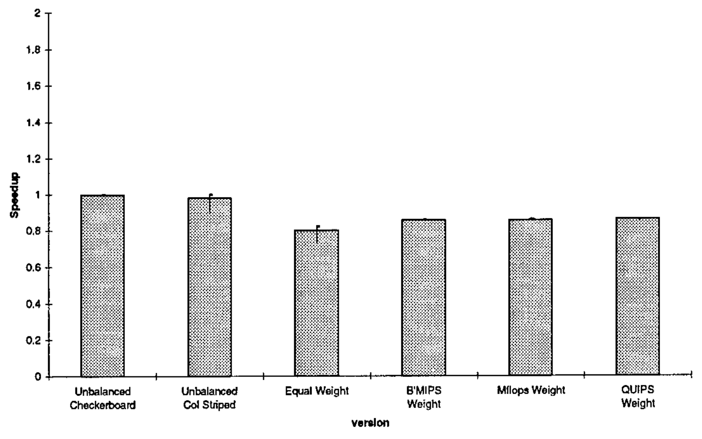


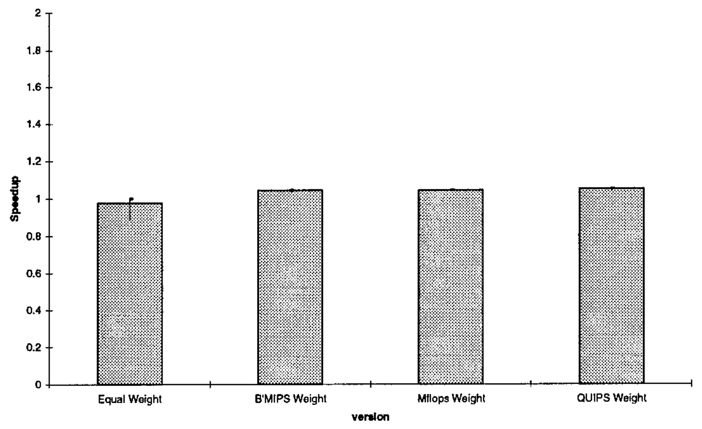
Figure 4-5. Performance with & without load balancing – 2 processors, 1x400 1x450.

4.3. Four-Processor Results

As with the two-processor tests, the four-processor tests made use of three combinations, in which the least powerful processor was a 200 MHz Pentium, a 333 MHz Pentium II, and a 400 MHz Pentium II, respectively. Like the uniprocessor tests, we found that the code with the block checkerboard partitioning was plagued with NaN exceptions.



(a)



(b)

Figure 4-6. Speedup over best non-load balanced performance – 2 processors, 1x400 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

4.3.1. 1x200 1x333 1x400 1x450

In the tests with one processor of each type, we observe (Figure 4-7) that the memory penalty is minimal, and that greater improvements are realized when load balancing is implemented. More specifically, the performance of the "equal weight" partitioning is only 3.3±0.47% less than the "unbalanced" performance.

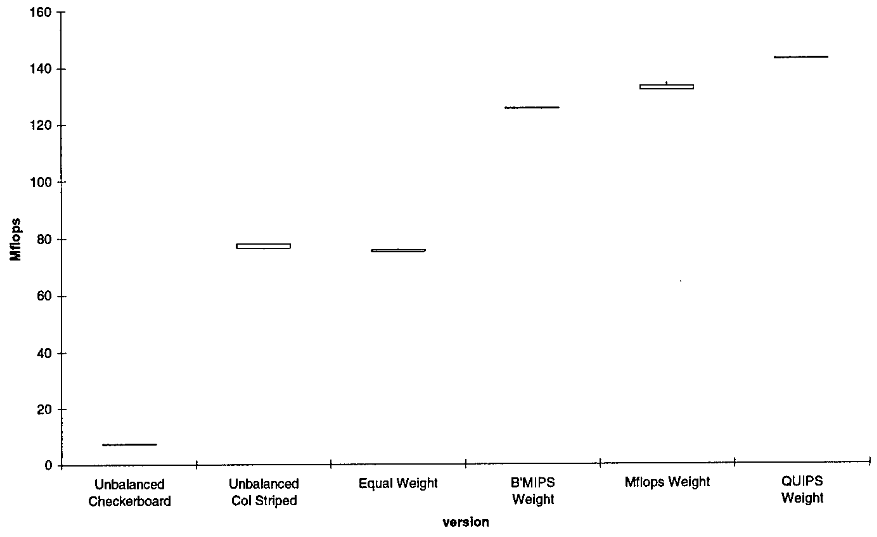
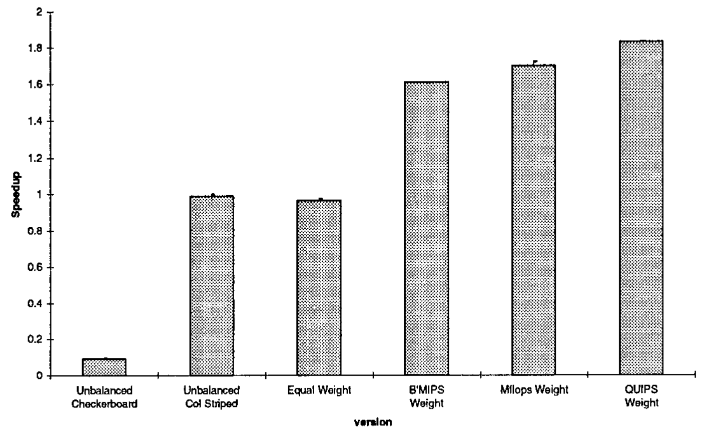


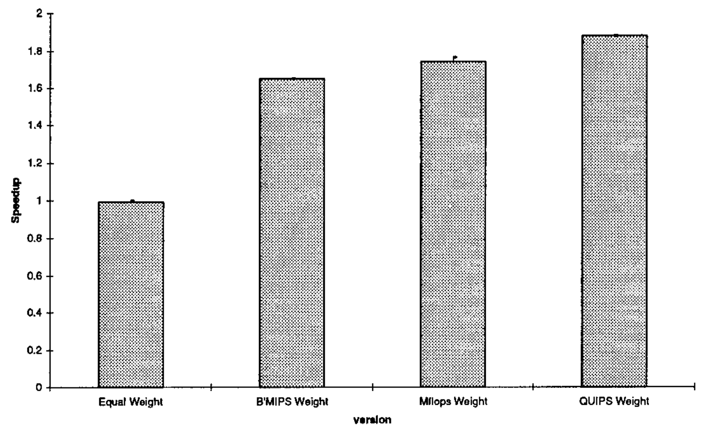
Figure 4-7. Performance with & without load balancing – 4 processors, 1x200 1x333 1x400 1x450.

The speedup over the unbalanced code (Figure 4-8) is 60.9±0.02% when weighing the nodes with BogoMIPS, 69.8±1.33% when weighing with Mflops, and 83.1±0.17% when using QUIPS. Accounting for the memory penalty, these numbers increase to 65.1±0.03%, 74.4±1.37%, and 87.9±0.17%, respectively.

As with the two-processor case with the Pentium, we ask whether the low-end processor contributes to the performance. Since the power-of-two processors requirement was removed, we were able to execute LU using three processors, using a combination identical to the combination discussed in this section, except that the Pentium was not used. The results of this test are in Table D-3.



(a)



(b)

Figure 4-8. Speedup over best non-load balanced performance – 4 processors, 1x200 1x333 1x400 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

Curiously, the performance of the code with the "memory penalty" is better than that for the code without the penalty. Given the tight variance for each[61] and the fact that the nodes were dedicated for these tests, it is difficult to attribute this reversal to competition for the processors. Nevertheless, the best performance for the three-processor case is 134.9 Mflops, using QUIPS-weighted load balancing. Using QUIPS-

[61] σ^2=0.00297 Mflops^2 & σ^2=0.02052 Mflops^2, respectively.

weighted load balancing, the four-processor case with the Pentium outperforms the best three-processor performance by 7.9±0.13 Mflops, or 5.9±0.10%.

4.3.2. 1x333 2x400 1x450

The next tests replace the Pentium with a 400 MHz Pentium II; the results are shown in Figure 4-9. As with the two-processor tests without the Pentium, the performance improvement with load balancing is not appreciable.

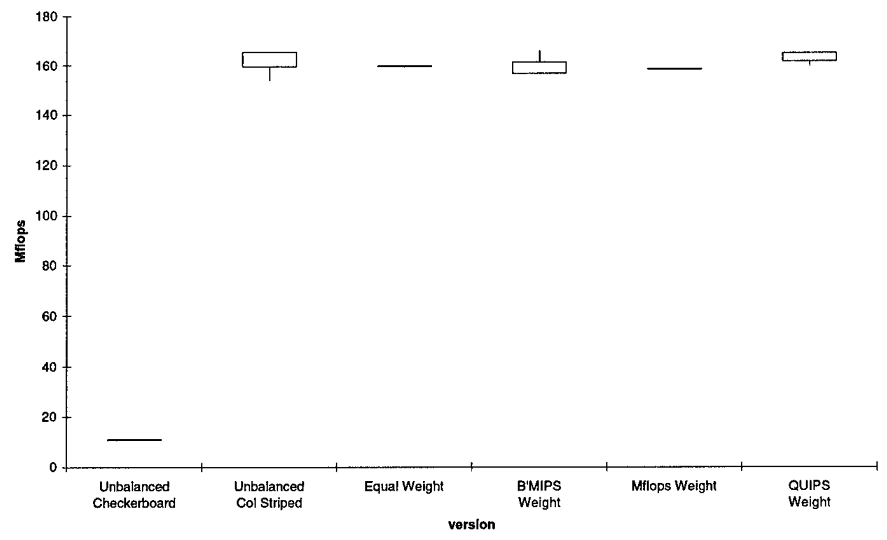
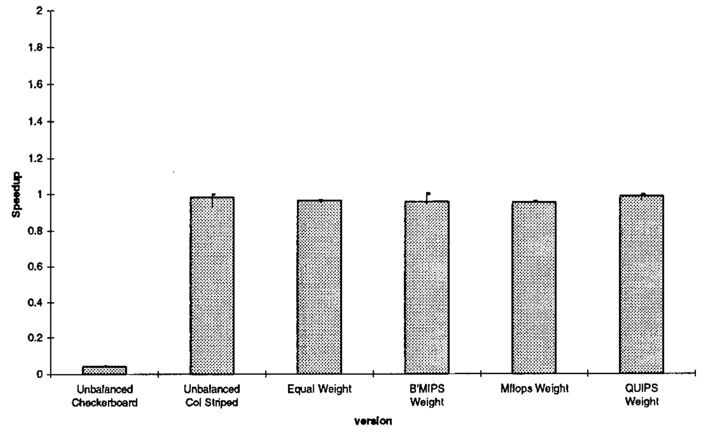
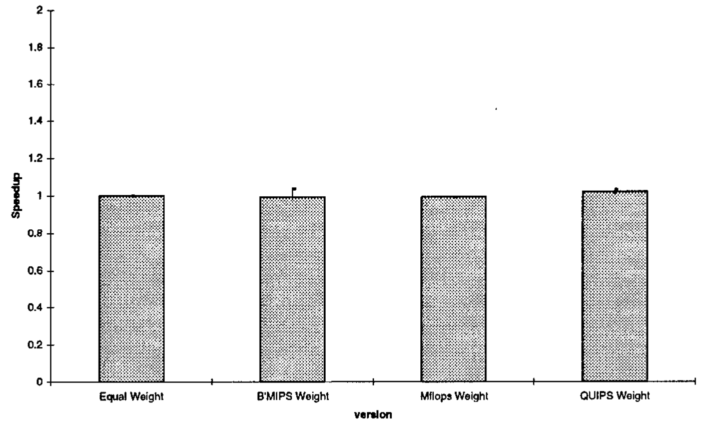


Figure 4-9. Performance with & without load balancing – 4 processors, 1x333 2x400 1x450.

The memory penalty for this combination is also low, 3.5±0.14%. The load balancing schemes do not show an improvement over the unbalanced code (Figure 4-10), except for one instance in which the load balanced code exceeds the best unbalanced performance by 0.2%. Even comparing against the best equally-weighted code with the memory penalty, the best load balanced performance is a 3.7% improvement; only the QUIPS weighting provides an improvement of 2.2±1.38%. No value from the Mflops weighting is greater than the best equal-weighted performance, and the Wilcoxon test suggests a 69% probability that the true BogoMIPS-weighted performance is less than the best equal-weighted performance.



(a)



(b)

Figure 4-10. Speedup over best non-load balanced performance – 4 processors, 1x333 2x400 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

4.3.3. 3x400 1x450

The final four-processor combination uses three 400 MHz Pentium IIs and a 450 MHz Pentium II, with the results shown in Figure 4-11.

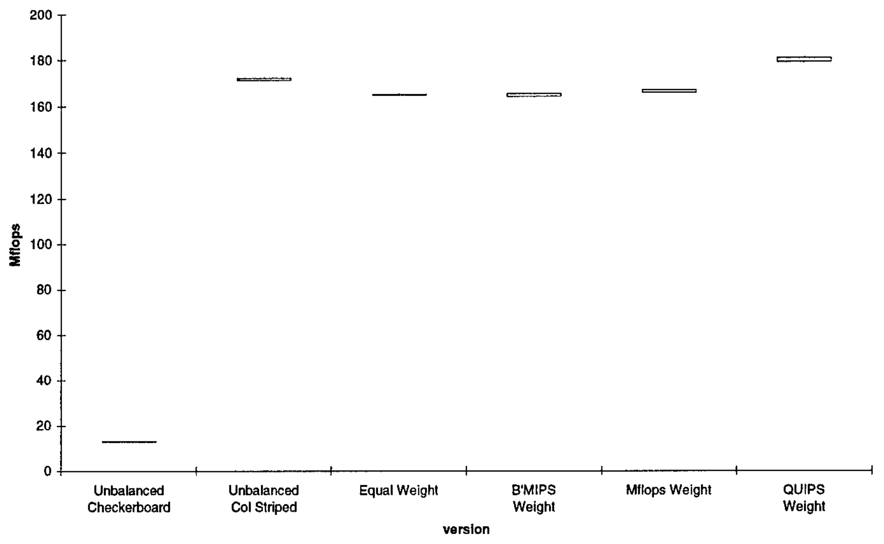
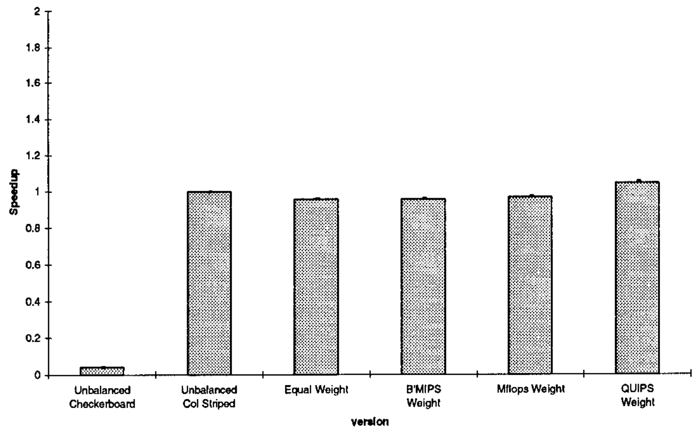


Figure 4-11. Performance with & without load balancing – 4 processors, 3x400 1x450.

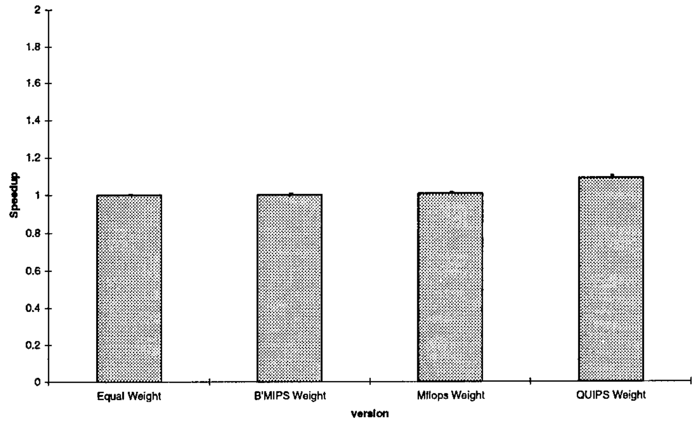
The penalty for overallocating memory is somewhat greater than it is in the other four-processor cases, 4.3±0.11%. Only with QUIPS weighting does the load balancing outperform the unbalanced code (Figure 4-12), by 4.3±0.54%; QUIPS weighting outperforms equal weighting by 8.8±0.56%, and Mflops weighting does by 0.8±0.37%.

4.4. Eight-Processor Results

The last set of tests make use of eight processors. There are two such combinations, one with the least powerful processor a 200 MHz Pentium, and one with the least powerful processor a 333 MHz Pentium II. Unlike the two- and four-processor tests, there are no eight-processor tests using only 400 MHz & 450 MHz Pentium IIs, as there are not eight such processors in the ABC.



(a)



(b)

Figure 4-12. Speedup over best non-load balanced performance – 4 processors, 3x400 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

4.4.1. 1x200 1x333 5x400 1x450

In Figure 4-13 we observe that, as with the other experiments which use the Pentium, the load balanced code shows considerable speedup over the unbalanced code. We also notice a greater range of values than was present in the tests with fewer processors.

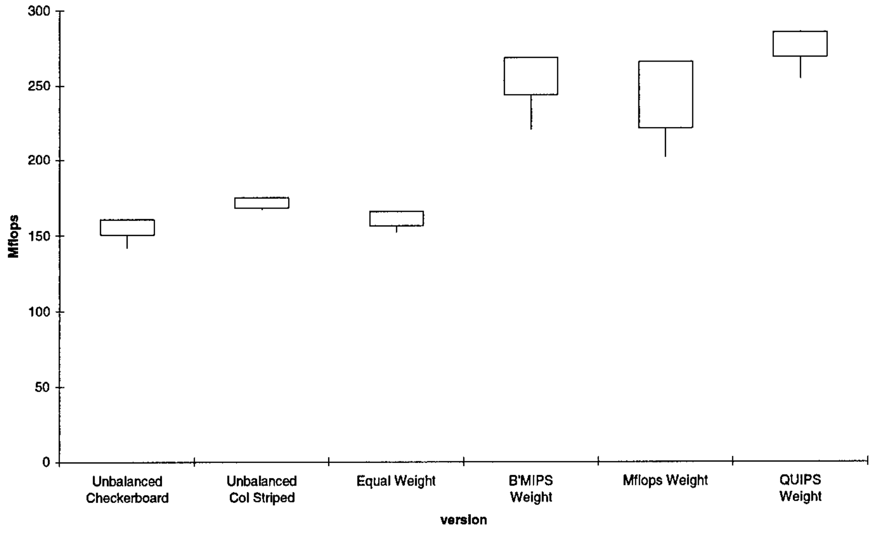
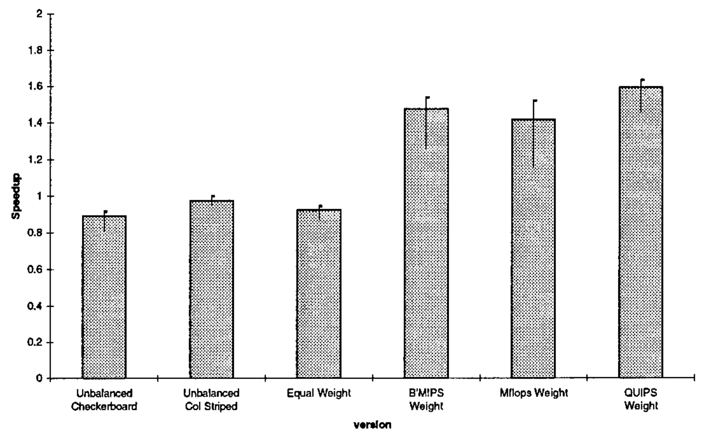


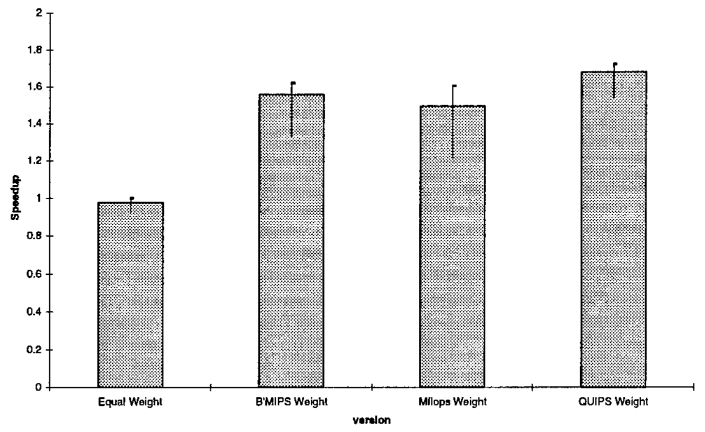
Figure 4-13. Performance with & without load balancing – 8 processors, 1x200 1x333 5x400 1x450.

We observe in Figure 4-14 that the BogoMIPS weighted code has a performance improvement of 47.4±12.2% over the best unbalanced performance; speedup with the Mflops weighting offers 41.4±16.0%; and QUIPS weighting 58.9±7. 7%. Compensating for the 7.7±3.2% memory penalty, and the three weighting schemes provide respective improvements of 55.8±12.9%, 49.5±16.9%, and 68.0±8.1%.

Once again, we check to determine if including the Pentium is better than using the same processors except the Pentium. The results of the seven-processor runs are in Table D-3. We find that the worst unbalanced seven-processor performance is still better than the best balanced eight-processor performance, likely due to the memory penalty.



(a)



(b)

Figure 4-14. Speedup over best non-load balanced performance – 8 processors, 1x200 1x333 5x400 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

Using one-tailed Wilcoxon signed rank tests, we determine that there is a 68.9% probability that the QUIPS weighted eight-processor code has a greater performance than the equal weighted seven-processor code with memory penalty. However, if the seven-processor code is also load-balanced, then we find there is a 96.9% probability that the QUIPS weighted seven-processor code outperforms the QUIPS weighted eight-processor code.

4.4.2. 1x333 6x400 1x450

The performance LU realized when using eight Pentium IIs is provided in Figure 4-15. Casual observation reveals there is a noticeable penalty for allocating too much memory and that only the QUIPS weighted partitioning offers performance benefits.

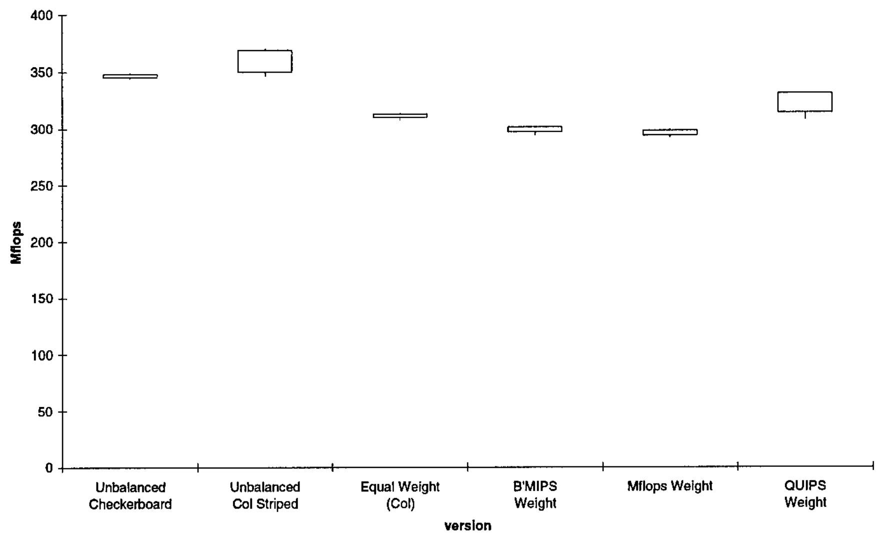
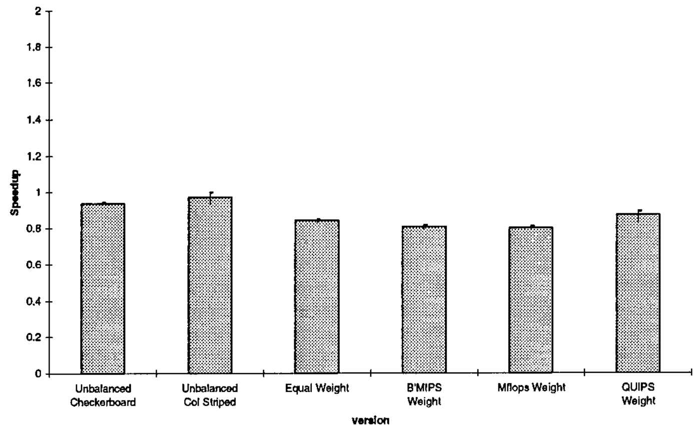
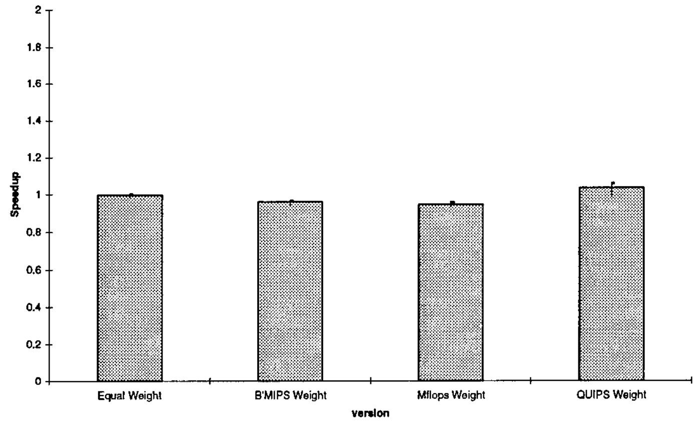


Figure 4-15. Performance with & without load balancing – 8 processors, 1x333 6x400 1x450.

Figure 4-16 reveals that not a single instance of the load balanced code could outperform the unbalanced code. Even taking the 15.8±0.61% performance loss due to the memory penalty, neither BogoMIPS nor Mflops weightings can offer performance improvements. Only QUIPS does, with a 0.0625 level of significance, offering a 3.4±3.2% improvement.



(a)

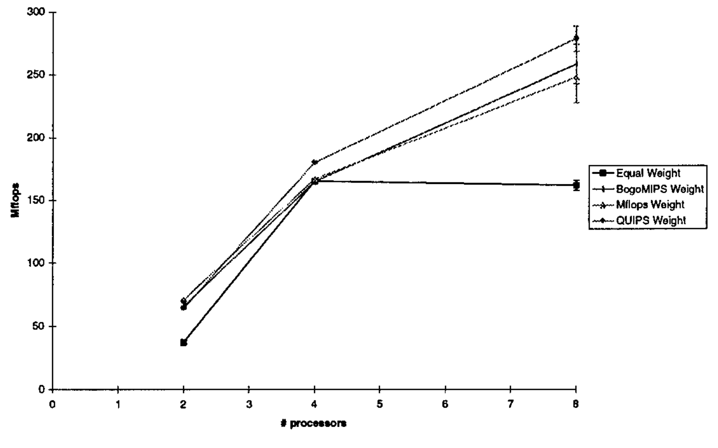


(b)

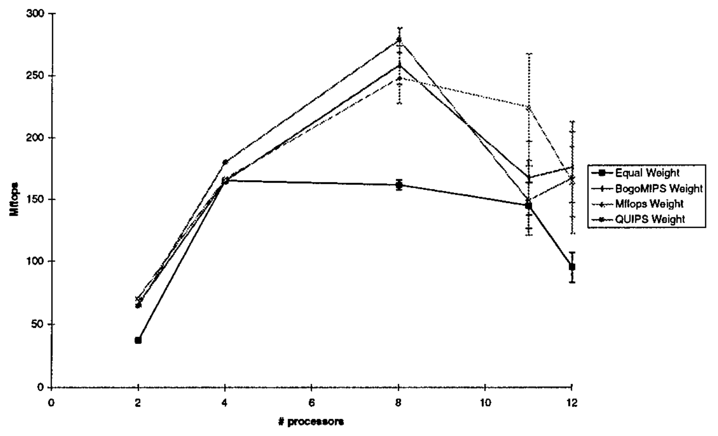
Figure 4-16. Speedup over best non-load balanced performance – 8 processors, 1x333 6x400 1x450. (a) Compared to best unbalanced time. (b) Compared to best unbalanced time with memory penalty.

4.5. Performance Beyond Eight Processors

A secondary goal in this thesis effort is to characterize LU's performance in megaflops when using the entire ABC. Figure 4-17 shows the performance as a function of the number of processors, when both the 450 MHz Pentium II and the 200 MHz Pentium are included in the processors. Figure 4-18 shows the performance when the fastest combination of processors are used for each quantity.



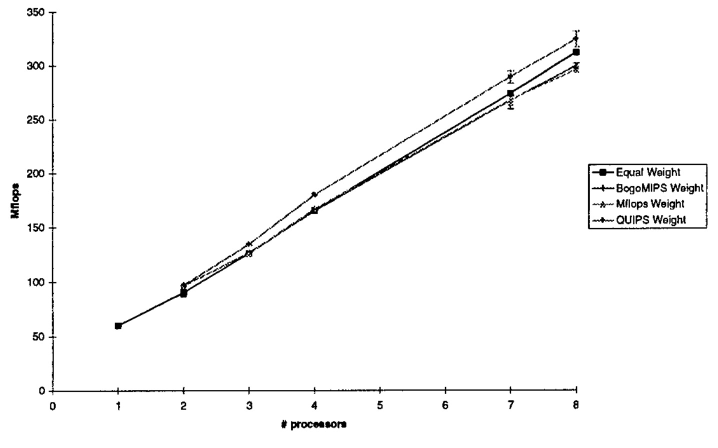
(a)



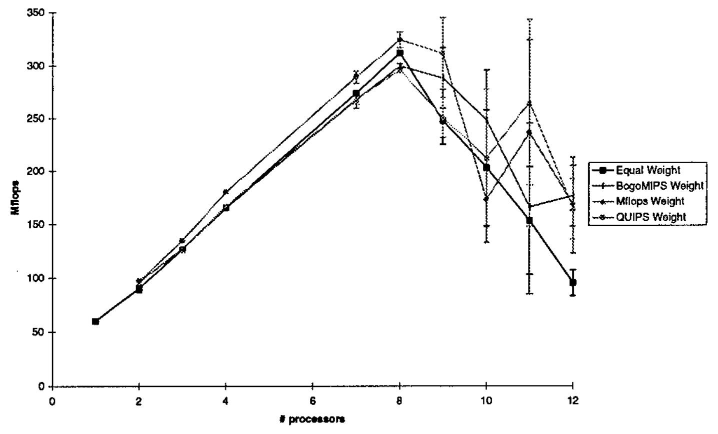
(b)

Figure 4-17. Performance in megaflops as a function of number of processors: broadest combination of processors.
(a) Up to network saturation. (b) Up to & past network saturation.

Figure 4-17a shows continuous performance improvements when using load balancing, up to eight processors. Similarly, Figure 4-18a shows steady speedup, both with and without load balancing, though QUIPS weighted load balancing does show better performance than no load balancing.



(a)



(b)

Figure 4-18. Performance in megaflops as a function of number of processors: fastest processors. (a) Up to network saturation. (b) Up to & past network saturation.

Figure 4-17b and Figure 4-18b show that beyond eight processors, performance decreases and the variance of the measurements increases dramatically. The explanation is that the ABC's ICN switch does not have unlimited capacity. Its internal capacity may aggregate to 6.3 Gbps, but the effective network capacity is 800 Mbps [45:78]. Since each process attempts to exchange information at the same time the other processes do, network collisions become inevitable when more than eight processes are used; nine or more nodes sending information at 100 Mbps each exceed the effective network capacity of the switch.

4.6. Other Observations

4.6.1. Not-a-Number Exceptions

In the one- and four-processor cases, the code which uses block checkerboard partitioning consistently generates NaN exceptions at some point during the solution, which then propagate through to the rest of the solution. Handling all these exceptions severely reduces the performance, to say nothing of the correctness of the solution.

As can be seen in Table D-19, this phenomenon did not always occur. Further, no one else has reported such a problem [100]. We are unable to determine the source of these NaNs. The binaries used during software development and during the experiments were generated with the same compilers using the same options and the same libraries on the same systems. During development, the code executed without exceptions using both the hub and the switch, which eliminates the network as a potential source of the problem. During experimentation, we executed the code on four different nodes, and NaNs were generated on all four nodes, leading us to conclude that a single node is not the source of the NaNs.

4.6.2. Effect of Non-Unit Stride Memory Access

To quantify the performance penalty of using rowwise striped partitioning,[62] we executed LU coded for rowwise striped partitioning on eight processors, including the 200 MHz Pentium (Figure 4-19). What we find is that, as expected, there was not a

[62] See Section 3.3.2.2.

performance penalty if the correct amount of memory is allocated for each tile. However, for the "equal weight" case, in which extra memory was allocated for load balancing, the effect is that the Pentium node, with its 32 MB of memory, must constantly swap data in and out of virtual memory, reducing the overall performance by two orders of magnitude.

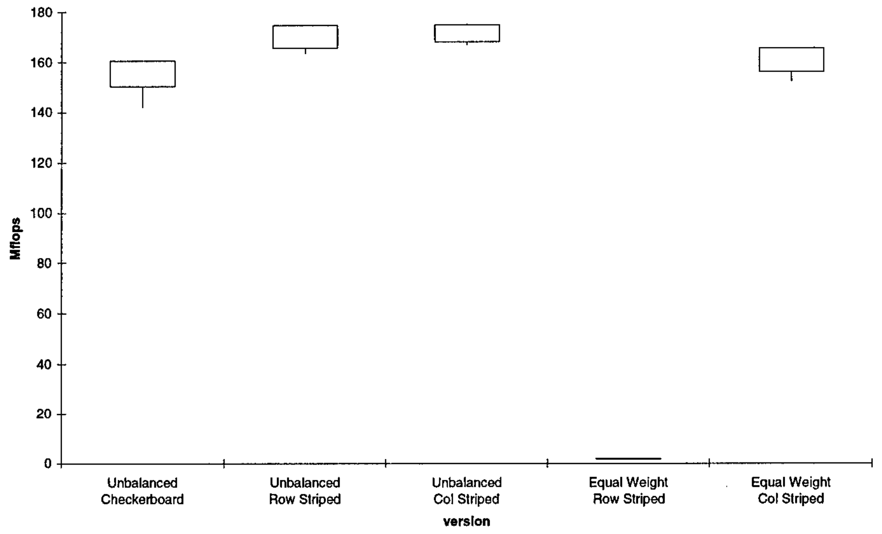


Figure 4-19. Performance with rowwise striped & columnwise striped partitioning – 8 processors, 1x200 1x333 5x400 1x450.

4.6.3. Price-Performance

In determining a price-performance ratio, we must determine both the price of the system and its specific performance. In the preceding sections, we discussed the performance of the ABC when running the LU application.

To estimate the cost, we shall make three assumptions. First, each node has a purchase price of $2000 when new, and the switch has a purchase price of $2500 new. Second, the nodes' monetary value depreciates at a rate of 33% per year; the switch does not depreciate. This depreciation is calculated for the time during which the experiments were conducted. Third, for this analysis, only the costs involved in the Linux portion are considered,[63] and licenses for software not available for this thesis are not considered, either.[64] This brings the system price to $21,664, as shown in Table 4-1.

Table 4-1. Price of the AFIT Bimodal Cluster (Linux).

	Quantity	Purchase Price	Purchase Total	Age	Depreciated Price	Depreciated Total
200 MHz Pentium	1	$2,000	$2,000	19 months	$1,061	$1,061
333 MHz Pentium II	4	$2,000	$8,000	11 months	$1,386	$5,544
400 MHz Pentium II	6	$2,000	$12,000	7 months	$1,584	$9,504
450 MHz Pentium II	1	$,2000	$2,000	3 month	$1,810	$1,810
Switch	1	$2,500	$2,500	—	$2,500	$2,500
Software	1	$20	$20	—	$20	$20
			$26,520			$20,439

We notice that the best uniprocessor performance provides a price-performance ratio of $34/Mflop at the "new" price and $31/Mflop[65] at the depreciated price when only that uniprocessor and the software is considered in the price. This demonstrates a potential problem with the "price-performance" statistic, namely an implied assumption that the objective is to obtain the best price-performance ratio when we run high performance applications.

While we may be interested in achieving high performance computing at commodity prices, once we have the system, our objective is to obtain the best performance. In this case, the best performance is 370.35 Mflops using eight Pentium IIs

[63] While the Windows NT license is included with most of the nodes at purchase, there are other licenses for software used with Windows NT that do not impact this thesis, such as compilers and MPI implementations.
[64] Specifically, the price of the Fortran 90 license is not included.

(Section 4.4.2). When determining the price to achieve this performance, we can determine the price one of three ways:

a) Use the price of just the eight nodes, the switch, and the software,

b) Use two-thirds of the price of the entire system, or

c) Use the price of the entire system.

The price-performance ratio using each of these approaches is shown in Table 4-2:

Table 4-2. Price-Performance Ratio.

	Purchase Price		Depreciated Price	
	Price	Price-Performance	Price	Price-Performance
Option (a)	$18,520	$50/Mflop	$15,220	$42/Mflop
Option (b)	$17,680	$48/Mflop	$13,626	$37/Mflop
Option (c)	$26,520	$72/Mflop	$20,439	$56/Mflop

Since we are attempting to assess the price-performance ratio for the best performance achieved using the system, the best answer is the one which considers the price of the entire system, $56/Mflop.

4.6.4. Comparison with Other Platforms

To place our best performance of 370.35 Mflops with eight processors in perspective, we compare this to the performance of other platforms executing the LU application, reported in [99]; this comparison is summarized in Figure 4-20a. An IBM SP (66/WN) was able to obtain 457.8 Mflops using eight processors, and an SGI/Cray T3E-1200 reached 610.8 Mflops with eight processors. Clearly, the ABC's performance is not on par with commercial supercomputers, but we expected this, particularly due to the ABC's commodity ICN.

[65] Rounding up to the nearest dollar.

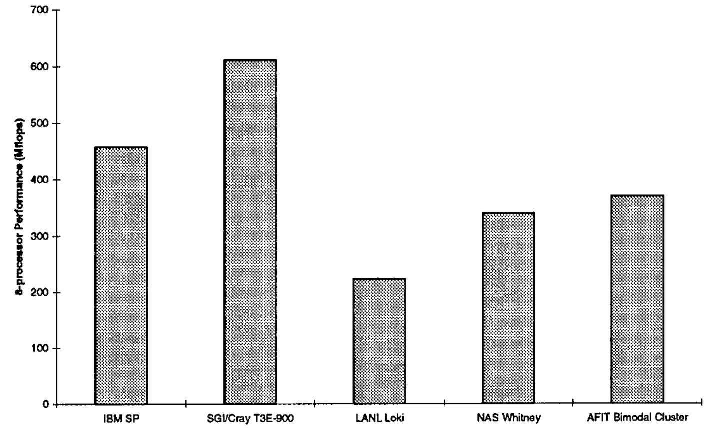
So how does the ABC compare with other commodity clusters of PCs? The results for Los Alamos National Laboratory's Loki and NAS' Whitney clusters are also of interest. For eight 200 MHz Pentium Pro processors, the older Loki cluster obtained 222.3 Mflops, and Whitney obtained 338.8 Mflops using eight 200 MHz Pentium Pro processors.

Pricing for the IBM SP is on the NAS website [99], and the pricing for the SGI/Cray T3E [80], Loki [97], and Whitney [88] are available on their respective websites. This provides us with sufficient information to make a price-performance comparison, summarized in Figure 4-20b.

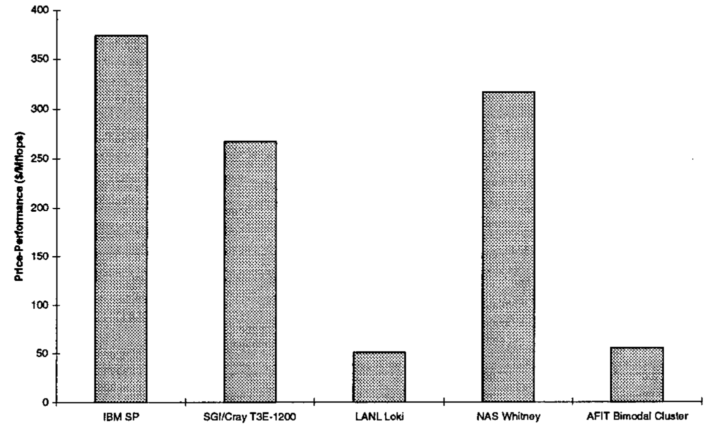
Using the same depreciation rules as for the ABC the current of the 64-processor IBM SP (66/WN) is just over $1 million, and its performance is 2.68 Gflops. SGI lists the price of a 32-node T3E-1200 as $630,000, and its 32-processor performance is 2.36 Gflops. The current price of the 16-node Loki is $24,683. Its best reported performance for LU is 453.0 Mflops using all 16 processors, yielding $55/Mflop. The 42-node Whitney, interconnected with Fast Ethernet and Myrinet, has a current price of $141,330, and its 32-processor performance with LU is 418.8 Mflops. This means the price-performance for Whitney is $338/Mflop, using the rule that the price of the entire system is used.

4.7. Analysis & Summary

In this chapter, we discuss the performance achieved using two, four, and eight processors, using combinations of processors that have a broad range of capabilities and that have a narrow range of capabilities, and using different weighting methods for our asymmetric static load balancing.



(a)



(b)

Figure 4-20. Comparison of systems, using LU.A. (a) Eight-processor performance. (b) Price-performance.

We find that when the range of capabilities is broad, all three weightings provide performance improvements over the unbalanced code. When the processors are all nearly equal in computational power, performance improvements are seen only after accounting for the performance penalty due to allocating more memory than is necessary to hold the partitions. Even then, there is not always a net improvement. We also find that when there is a broad range of processor capabilities, the weakest processor slowed down the arrival to the solution when using symmetric load balancing. But asymmetric load balancing permits all processors to contribute to the solution; the exception is the eight-processor case, in which the Pentium processor lessened the performance of the system even with asymmetric load balancing.

We observe that in six of the eight combinations of processors considered, the QUIPS weighting provides the best load balancing. Exactly why this should be so is not entirely clear, though, as in four of these six cases, the QUIPS weighting produces the same partitioning as one or both of the alternate weighting schemes that QUIPS outperforms. We speculate this may be due to the extra code associated with the `run_hint()` function affecting memory alignment, but without further investigation, we can not make any such statement with certainty.

Further discussion and conclusions that can be drawn from these results are presented in the next chapter.

V. Conclusions & Recommendations

As a result of this thesis effort, students and faculty at AFIT have ready access to a low-cost high performance computing platform for their research, software and techniques are available for those researchers to make more efficient use of this system, and they can obtain this effective load balancing with less overhead than other static assymetric load balancing approaches.

But there is still more work that can be done in this field of research. The results & analysis presented in Chapter IV lead us to certain conclusions about the load balancing algorithm developed in this thesis research. We also make recommendations for future work with asymmetric load balancing and for the continuing development of the AFIT Bimodal Cluster.

5.1. Load Balancing Conclusions

After correcting for the penalty imposed due to overallocating memory, we find that the QUIPS rating consistently provides better performance than the unbalanced code, regardless of the range of processor capabilities, up to eight processors. If the range of processor capabilities is sufficiently wide, then all three weighting techniques provide an improvement over the unbalanced code.

We also have determined that for the two- and four-processor cases, the load balancing allows us to make full use of the available processors; if load balancing were not used, we may realize better performance by leaving out the weakest processor. Why doesn't load balancing with the eight-processor case permit us to make full use of the processors? The answer lies in the last row of Table D-12. Using the weights returned by the NodeMetric library, ABC12's "fair share" of the problem is a tile two or three elements wide. The load balancing algorithm is designed to reshift the balance to prevent any processor from having a tile less than four elements wide. So ABC12 is still overtaxed, while ABC03 is not being used to its fullest extent. If this requirement were removed, or if we were using a larger problem size, then the partition reshifting would be unnecessary. Each processor would still be responsible for its fair share, and we would see a performance improvement over the seven-processor case.

We also observe that we have reduced the time needed to make use of the HINT benchmark, when compared to the initial approach [82]. To build the maps used by the metricmap functions, just over forty-three hours of processor time was used to build maps for five intrinsic data types.[66] Had we been required to execute the HINT benchmark on *every* node, even if only for the double-precision floating point version, then just over fifty-one hours of processor time would be required.[67] Further, when new nodes are added to the cluster, we are not first required to execute the HINT benchmark on them, unlike the approach in [82].

5.2. *Future Asymmetric Load Balancing Efforts*

The situation with comparing seven processors to eight, in which the 200 MHz Processor does not contribute to a faster solution, as described previously, forces us to reexamine why we do not permit tiles to be narrower than four elements. We suspect the

[66] 528.90 min on 200 MHz Pentium; 655.95 min on 333 MHz Pentium II; 628.20 min on 400 MHz Pentium II; and 770.48 min on 450 MHz Pentium II.
[67] 172.87 min on 200 MHz Pentium; 277.63 min on each 333 MHz Pentium II; 266.95 min on each 400 MHz Pentium II; and 176.85 min on 450 MHz Pentium II.

original reason for this constraint was to prevent the use of so many processors that interprocessor communication destroyed the performance. So, the real constraint is not a lower limit on the size of the tiles, but rather an upper limit on the number of processors that may be used. So long as we are not exceeding this upper limit, then there is no reason why the weakest processor cannot be responsible for a tile narrower than four elements. So long as its “fair share” is a tile at least one element wide, there is no reason why the weakest processor could not contribute to the solution when it is not tasked with more than its “fair share.” For this reason, the lower limit on the width of the tiles could be removed in the load balanced code.

A Fortran 90 compiler does not have to overallocate memory to permit asymmetric load balancing. Instead, it can dynamically allocate memory at run-time, after the partition sizes have been determined. These tests should be run again after being compiled with a Fortran 90 compiler, to establish the effects of changing compilers, and then the application should be modified to make use of dynamic allocation. That accomplished, the tests should be run yet again to ascertain the effect on performance that asymmetric load balancing with dynamic memory allocation has. Once dynamic memory allocation is used, the B-class problem can be executed on the 8-processor combinations: ABC12 should never be required to allocate more than 20.24 MB for a partition when 8 processors are used to tackle the B-class problem, and this fits inside ABC12’s main memory.

An alternate approach that should be tried is “diffusive load balancing,” described by Corradi, *et.al.* [17]. In diffusive load balacing, the workload is shifted between neighboring processors if there is a load imbalance between them; gradually, a global balance is achieved. Using diffusive load balancing, it should be straight-forward to achieve an asymmetric load balance *without weighing the nodes*. If the time a processor spends in an MPI_Wait() call is above some threshold, then diffusive potential is indicated.

After these suggested changes have been implemented and tested, two-dimensional load balancing also should be attempted, as described in Section A.1.3.3.

5.3. Development & Future Directions for the AFIT Bimodal Cluster

The author constructed the ABC and has managed it for nearly a year. This thesis, along with those of other students, has made use of this expanding high performance computing platform. The ABC should continue to grow to permit research into larger problems and to accommodate a greater number of researchers. Hand-in-hand with the ABC's growth is the need for the hiring of a system administrator to manage the cluster. Even with twelve nodes, being able to effectively serve as a system administrator is growing beyond the time requirements for full-time graduate students.

Scaling the interconnection network with the cluster may prove challenging, but the issues involved and solutions are discussed in [77]. The primary considerations are the latency of an individual message and the channel capacity. The authors of [77] suggest a tree of switches with Fast Ethernet leaves uplinked to a Gigabit Ethernet router as the best option considered. While they do not explicitly address the use of very large Fast Ethernet switches (such as by stacking multiple 24-port switches), we recommend that a tree structure is more appropriate, since it isolates network traffic on the leaf switches from the traffic on the other leaf switches (except where the traffic *must* cross the Gigabit Ethernet link to another leaf switch), and thus should provide superior performance. A network simulation is needed to provide a quantitative case for a tree of switches versus stacked switches.

The Linux kernel needs to be replaced or repaired. The bug in the `tcp_ack()` function can be fixed by modifying the kernel's source code and recompiling [76], but the TCP stack in the 2.2.x kernels has undergone several improvements [18]. This, combined with Linux 2.2.x's improved performance on systems with more than 16 MB of main memory [18] suggest upgrading the kernel is the wiser solution.

Finally, some updated development software should be implemented. MPICH 1.1.2 [9] is available for beta testing now, and it is supposed to fix some errors in previous versions of MPICH 1.1.x. We also recently learned of the Pentium Compiler Group [65] which has developed patches (pgcc) to the egcs compiler suite to provide optimizations particular to Intel processors at the Pentium level and newer. Making use of pgcc compilers should offer greater performance over our current egcs compilers, as the egcs compilers are general compilers for any 32-bit processor that implements the Intel Architecture.

5.4. Closing Thoughts

This thesis effort has not attempted to prove or demonstrate that clusters of PCs are more effective or more efficient than MPPs, or even that PoPCs are sufficient replacements for some applications. Instead, it starts with the premises that PoPCs are already here in numbers, and that they are an inexpensive way to execute high performance computing applications.

From that premise, the issue this thesis addresses is how to make more efficient use of these clusters. Many of the traditional assumptions about supercomputing platforms do not hold true with commodity clusters, particularly when we realize that PoPCs have certain grown potentials that are not possible with the "big iron" machines, particularly the ability to add the most "powerful" processors to the system as money become available, rather than limiting growth to the addition of more processors identical to those already in place.

The experiments that this document records have shown that with proper load balancing, computational scientists and engineers using PoPCs can efficiently use both the newest hardware in the system and the oldest, without the older hardware limiting the system's performance. In so doing, we conclude that the removal of older hardware is unnecessary when the newer hardware has more than twice the performance. Researchers are then able to get more use out of their research dollar, and obsolescence of the older hardware is delayed.

Appendix A: Supplemental LU Background Material & Analysis

A.1. Data Partitioning

There are several ways a matrix can be partitioned among processors, though only the three relevant to this thesis effort are addressed here. They are block checkerboard partitioning, rowwise block striping, and columnwise block striping. These are two-dimensional partitioning schemes; however, if the designers choose not to partition the *z* axis, as is the case with the LU application[68] [61], then two-dimensional partitioning schemes are suitable.

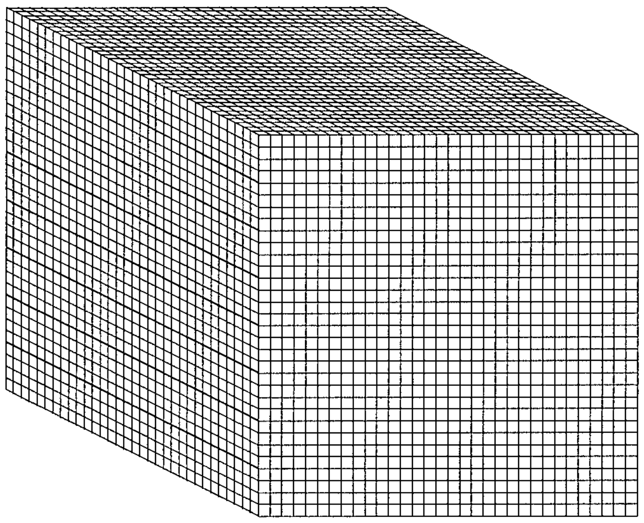


Figure A-1. Unpartitioned data set.

Detailed discussion on block checkerboard and block striped partitioning follows and is summarized in Table A-1:

[68] See Section 2.3.1, [10], or [101].

Table A-1. Features of Checkerboard and Striped Partitioning.

	Block Checkerboard Partitioning	Block Striped Partitioning
Primary Characteristic	Decomposes domain along two axes	Decomposes domain along one axis
Maximum Number of Processors	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Limitations on Number of Processors	Must be non-prime. Typically, power-of-two or square.	No restrictions for quantities less than maximum.
Communication Overhead	Less overhead for greater numbers of processors.	Less communication overhead for small numbers of processors.
Ease of Load Balancing	Difficult, but not impossible.	Easier, but not trivial.

A.1.1. Checkerboard Partitioning

In block checkerboard partitioning, the application partitions each *x*-*y* plane into smaller square or rectangular blocks (or "tiles") by partitioning the planes along both the *x* axis and the *y* axis [48:152]. The exact method varies from application to application.

Given p processors partitioning an $i \times j$ matrix, and two factors, p_1 and p_2 such that $p_1 \times p_2 = p$, each plane is partitioned into $p_1 \times p_2$ tiles with approximate dimensions $i/p_1 \times j/p_2$. These dimensions are approximate because, if i and j are not evenly divisible by p_1 and p_2, then some rounding may be required. As a specific example, if p is square and the matrix is an $n \times n$ square, then $p_1 = p_2 = \sqrt{p}$, and the plane is partitioned into $\sqrt{p} \times \sqrt{p}$ square tiles, each approximately $n/\sqrt{p}$ on a side.

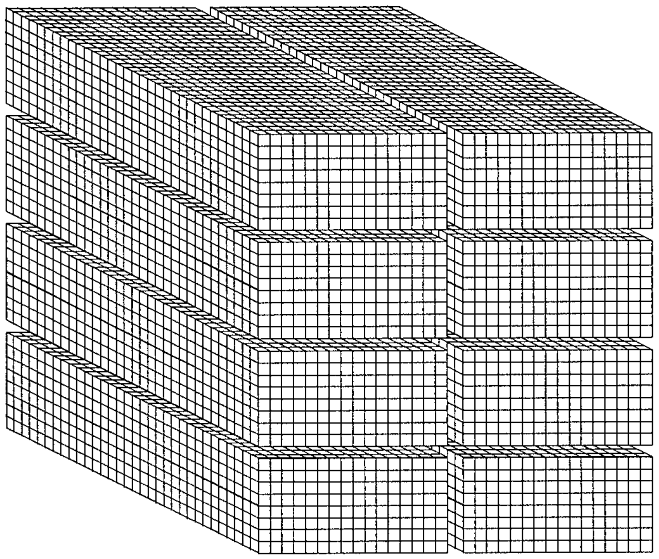


Figure A-2. Block checkerboard partitioning for eight processors.

As another example, the technique used by LU is to require that *p* be a power-of-two, and then alternately partition the *x* and *y* axes into greater powers-of-two, until all processors are utilized (Figure A-2).

A.1.2. Striped Partitioning

Block striped partitioning can be considered a special case of block checkerboard partitioning, in which one of *p*'s factors is 1. Alternately, if checkerboard partitioning is defined such that a processor is never assigned a complete row or column [48:152], then striped partitioning is a completely different mapping. Regardless, striped partitioning is a one-dimensional division of the matrix among the processors. We can either assign each processor full rows (Figure A-3) or full columns (Figure A-4) [48:151-152].

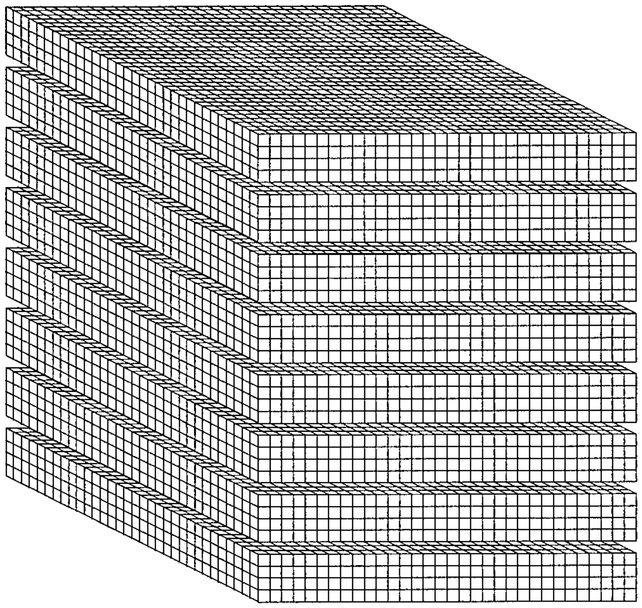


Figure A-3. Rowwise block spriped partitioning for eight processors.

From an abstract perspective, there is no fundamental difference between rowwise and columnwise partitioning; however, there may be performance reasons to select one over the other. These reasons may include minimizing processor idle time or minimizing communication overhead – and it may not be possible to do both. Another reason, particular to this thesis effort, has to do with how the implementation language stores matrices in memory; in this case, Fortran stores multidimensional arrays in column-major order [1:10].

A.1.3. Choosing a Partitioning Scheme

Why might a computational scientist/engineer select checkerboard partitioning over striping, or vice-versa? The two reasons most commonly cited are limits on the number of processors that can be used, and the communication overhead associated with each. In addition to these two, another reason relevant to this research is the ease by which the partitioning can be adjusted.

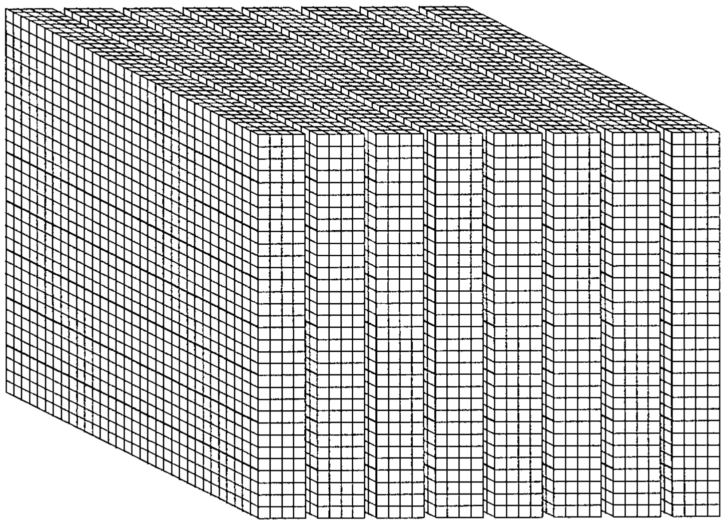


Figure A-4. Columnwise block striped partitioning for eight processors.

A.1.3.1. Number of Processors to be Utilized

A significant consideration is the number of processors amongst which the matrix can be divided. For an $n \times n$ matrix, striped partitioning can make use of $\mathcal{O}(n)$ processors, whereas checkerboard partitioning can utilize $\mathcal{O}(n^2)$ processors. As specific examples, consider LU's smallest and largest problem sizes. For the W-class problem, n=33, the problem can be divided among 64 processors using block checkerboard partitioning.[69] In contrast, the C-class problem specifies n=162, which makes the problem 188.3 times larger than the W-class problem;[70] yet with block striping, the application could only make use of 40 processors.

[69] The LU application requires that the minimum dimension on a partition be no less than four [61].

[70] $\frac{162^3}{33^3} \approx 118.3$

On the other hand, block striped partitioning does not constrain the number of processors used, short of the maximum. Block checkerboard partitioning schemes, at a minimum, require a non-prime number of processors. More typically, though, the number is more constrained (Table A-1). For example, the LU application requires a power-of-two number of processors; other applications such as SP require a square number of processors [10:8-9]. They can only make use of eight or nine processors out of the twelve available on the ABC. Looking forward, the ABC's switch can accommodate up to twenty-four nodes before we need to stack the switch with another;[71] when the ABC does have twenty-four processors, both LU and SP would only be able to use sixteen.

A.1.3.2. Communication Overhead

Another consideration is the communication pattern. This varies from application to application, so here the focus is on LU. If we assume a processor can send and receive at the same time, that network contention is not an issue, and that the time for a message to propagate through the network is constant, regardless of the sender and receiver, then the time required to send a message over the network can be expressed as

$$t_{comm} = t_s + t_w m \tag{A-1}$$

where t_s is the startup time required to prepare the message, t_w is the per-word transfer time,[72] and m is the number of words in the message; since the propagation time is constant, here t_s also includes the propagation time [48:45-48].

In the SSOR code, each tile exchanges data with each of its neighbors every iteration;[73] the typical tile has four neighbors in the block checkerboard case. If each

[71] See Section 3.1.

x-*y* plane has *n* elements on a side, there are *p* processors, and *n* is evenly divisible by $4\sqrt{p}$, then each tile has $n/\sqrt{p}$ elements on a side. To exchange the values of the border elements, then, requires sending $n/\sqrt{p}$ elements four times. An exchange requires the sending of five eight-byte words for each element [101:2]. Equation (A-1) then becomes

$$\begin{aligned} t_{comm,BC} &= 4(t_s + t_w m) \\ &= 4\left(t_s + \frac{5nt_w}{\sqrt{p}}\right) \end{aligned} \tag{A-2}$$

If LU were partitioned using block striping, then each tile has at most two neighbors. If *n* is evenly divisible by 4*p*, then each tile measures $n \times n/p$, and exchanging the values of the border elements requires the transmission of *n* elements twice, and equation (A-1) becomes

$$\begin{aligned} t_{comm,BS} &= 2(t_s + t_w m) \\ &= 2(t_s + 5nt_w) \end{aligned} \tag{A-3}$$

If $t_{comm,BC} < t_{comm,BS}$, then block checkerboard partitioning has a lower communication overhead for LU, and if $t_{comm,BC} > t_{comm,BS}$, then block striped partitioning has a lower communication overhead for LU. Algebraic manipulation leads us to

$$t_s > 5nt_w\left(1 - \frac{2}{\sqrt{p}}\right) \tag{A-4}$$

---

72 Defined as the inverse of the channel capacity in words per second.

73 See Section 2.3.1.

as the determination as to whether block striping has superior communication patterns. When using the ABC, the interconnection network has a capacity of 100 Mbps before the messaging overhead (which is represented by t_s). From this we can determine that

$$\begin{aligned} t_w &= \left(10^8\, {bits}/_{\sec}\right)^{-1} \\ &= 10^{-8}\, {\sec}/_{bit} \times {8bits}/_{byte} \times {8bytes}/_{word} \\ &= 6.4 \times 10^{-7}\, {\sec}/_{word} \end{aligned} \tag{A-5}$$

Combining equations (A-4) and (A-5), we have

$$t_s > 3.2n\left(1 - \frac{2}{\sqrt{p}}\right)\text{msec} \tag{A-6}$$

For the one-processor and two-processor cases, equation (A-6) compares t_s against a negative value, which always evaluates to be true. In point-of-fact, though, the partitioning is identical for block checkerboard and block striping in the one- and two-processor cases, and we would expect neither to be better. For the four-processor case, though, t_s is compared against zero, which also evaluates to be true. It stands to reason that in the four-processor case, block striping is superior, since the total number of words exchanged is the same for each scheme, but the messaging overhead occurs only half as often.

For greater numbers of processors, the balance point varies as a function of the messaging overhead, the number of processors, and the size of the problem.

A.1.3.3. Load Balancing

The remaining consideration, which that played a dominant role in this thesis effort, is the the ability to manipulate the size of the partitions. For block striping, the sizes can be manipulated by "merely" adjusting the dimension along the partitioned axis (Figure A-5b). For block checkerboard partitioning, the task is not as straight-forward.

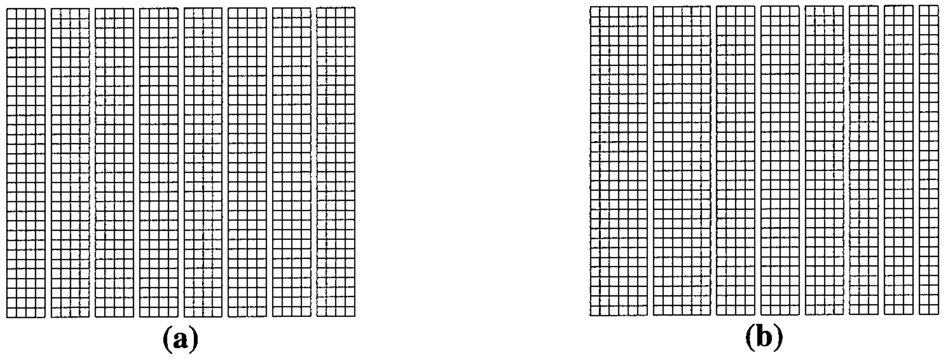


Figure A-5. Asymmetric load balancing using block striped partitions on the *x*-*y* plane. (a) Unbalanced. (b) Balanced.

One approach is to use block-checkerboard as a first estimate of the load balance, and then completely abandoning a clearly-defined partitioning scheme, making the tiles irregular shapes by adding and removing elements until each tile has an appropriate number of elements (Figure A-6b). The greatest problem here is that the entire CFD engine would have to be rewritten to accommodate the irregular shapes, both for computation and communication.

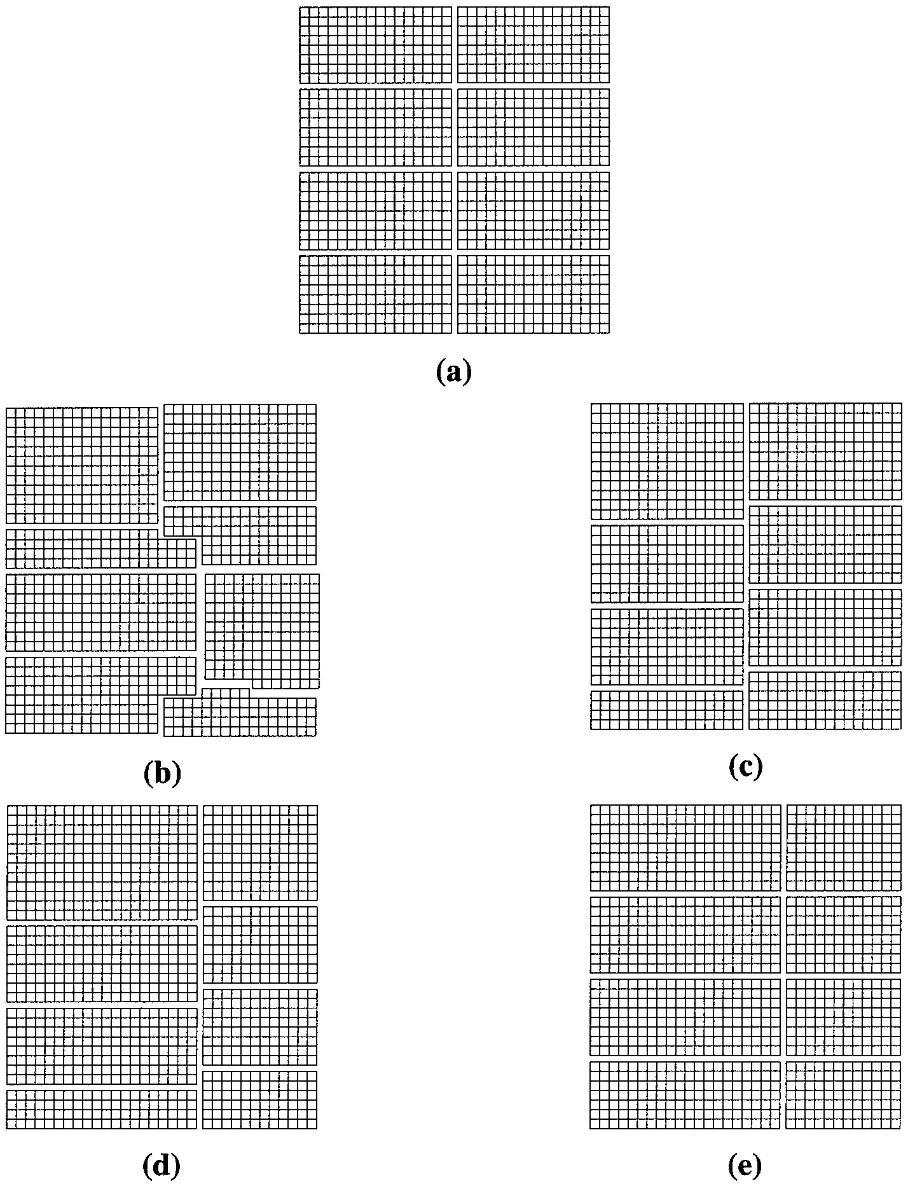


Figure A-6. Asymmetric load balancing using block checkerboard partitions on the *x*-*y* plane. (a) Unbalanced. (b) No fixed tile shape. (c) Fixed column width; row width varies within each column. (d) Variable column width; row width varies within each column. (e) Variable column width; row width varies globally.

The next approach that we might try is to fix the column widths, and within each column, adjust the row widths to achieve a balance in that column.[74] The most obvious problem is that this does not provide a global load balance. So we might use the aggregate capabilities of each column to adjust the width of the columns, and then adjust the row widths within each column, providing a better global load balance. The problem with both approaches in this paragraph is that they violate an assumption in the LU code, namely that each tile has at most one neighbor on each edge [61]. Overcoming this obstacle is not as difficult as rewriting the entire CFD engine, but it in the interest of incrementally modifying the application so as to improve our ability to isolate errors and unexpected behavior, we do not wish to make too many changes at once.

The final approach to load balancing a block checkerboard partitioned problem is to use the aggregate capabilities of each column to adjust the width of the columns, and then use the aggregate capabilities of each row to adjust the width of the entire rows instead of within each column. This is only somewhat more challenging than load balancing a block striped problem, but it does not provide as good of a load balance as is possible with the block striped partitioning. Nonetheless, if the communication overhead makes block striped partitioning undesirable, or if the number of processors to be used is greater than can be used with block striping, then block checkerboard partitioning with this last load balancing approach is the preferred technique.

[74] In this paragraph, the terms "row" and "column" may be reversed – it is more convenient to use these terms than "dimension *A*" and "dimension *B*."

A.2. Finite Difference Method

As might be expected, a system of partial differential equations must be discretized to be solved on a computer. The method briefly described here is the finite difference method (FDM). This is only a cursory treatment to aid the reader who is completely unfamiliar with computational fluid dynamics. Algorithms to solve the system of equations are not provided. Further, the description here is done in two dimensions for simplicity; the NPB LU simulated CFD application is a three-dimensional problem. A full treatment can be found in [7].

Table A-2. Advantages and disadvantages to higher-order accuracy with the finite difference method.

Advantages	Disadvantages
May require a smaller number of grid points to obtain a solution of the same accuracy, reducing the overall computation time.	Requires more compute time because there are more difference quotients to compute.
Often produces higher-quality solutions for certain scenarios.	Requires more compute time because each difference quotient requires more calculations than the lower-order difference quotients.
	By requiring access to grid points farther away from the grid point being evaluated, requires more communication between compute nodes when updating the boundary conditions between subdomains (not included in [7]).

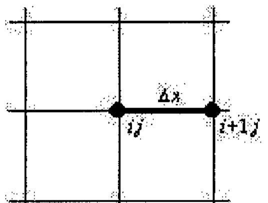
[7:128,132,135-137].

In the finite difference method, partial derivatives are replaced with algebraic difference quotients, or finite differences. Generally, this is based on Taylor's series expansions. A critical question is the degree of accuracy to be used in the expansion. Many consider first-order accuracy is insufficient for CFD applications. Second-order accuracy is considered quite sufficient for most CFD applications, though there are advantages and disadvantages to going to higher-order accuracy.

To consider how much the complexity of the FDM evaluation grows when going to higher-order accuracy, consider the growth from first-order accuracy to second-order accuracy in two dimensions. First, the first-order equations:

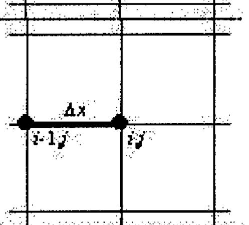
First-order forward difference with respect to x

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x}$$



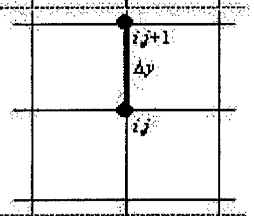
First-order rearward difference with respect to x

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x}$$



First-order forward difference with respect to y

$$\left(\frac{\partial u}{\partial y}\right)_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta y}$$



First-order forward difference with respect to y

$$\left(\frac{\partial u}{\partial y}\right)_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{\Delta y}$$

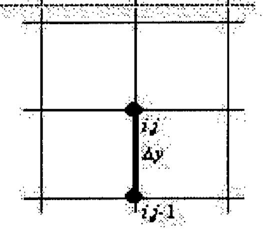


Figure A-7. First-order finite-difference expressions. [7:130-136]

Second-order accuracy makes use of the four first-order difference quotients and introduces five second-order difference quotients:

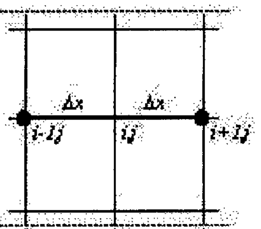
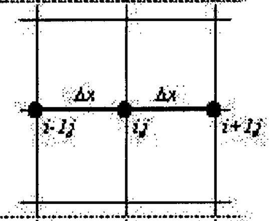
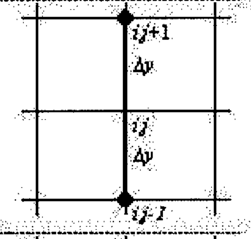
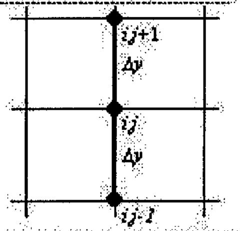
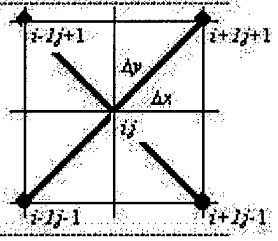
Second-order central difference with respect to *x*	$\left(\frac{\partial u}{\partial x}\right)_{i,j} = \frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}$	
Second -order central second difference with respect to *x*	$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} = \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\Delta x)^2}$	
Second -order central difference with respect to *y*	$\left(\frac{\partial u}{\partial y}\right)_{i,j} = \frac{u_{i,j+1}-u_{i,j-1}}{2\Delta y}$	
Second -order central second difference with respect to *y*	$\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} = \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{(\Delta y)^2}$	
Second -order mixed difference with respect to *x* and *y*	$\left(\frac{\partial^2 u}{\partial x\,\partial y}\right)_{i,j} = \frac{u_{i+1,j+1}+u_{i-1,j-1}-u_{i-1,j+1}-u_{i+1,j-1}}{4\Delta x\Delta y}$	

Figure A-8. Second-order finite-difference expressions. [7:130-136]

Clearly, increasing the accuracy dramatically increases the computational demands for a solution. The increased demand is not merely in the extra difference quotients that most be calculated, but in the increased complexity of the extra equations. The first-order terms each require two floating point operations, a subtraction and a division. In contrast, the second-order terms each require between four and seven floating point operations.

Extending to three dimensions makes the contrast even more dramatic: only two terms are added for first-order accurate solutions, as opposed to six additional terms[75] for second-order accurate solutions.

[75] Two first-order terms, and four second-order terms.

Appendix B: NAS Parallel Benchmarks -- Changes

The original source code is available from [99]. The modified source code is stored on the ABC in the `/home/cbohn/thesis` directory, and it is also available directly from the author (see Vita for contact information).

B.1. ***diff -r NPB-baseline NPB-mod0***

```
Only in NPB-baseline: BT
Only in NPB-baseline: CG
Only in NPB-baseline: EP
Only in NPB-baseline: FT
Only in NPB-baseline: IS
diff -r NPB-baseline/LU/Makefile NPB-mod0/LU/Makefile
7c7
< OBJS = lu.o init_comm.o read_input.o bcast_inputs.o proc_grid.o neighbors.o \
---
> OBJS = lu_wrapper.o lu.o init_comm.o read_input.o bcast_inputs.o proc_grid.o neighbors.o \
27a28,30
>
> lu_wrapper.o:       lu_wrapper.c
>       ${CCOMPILE} lu_wrapper.c
diff -r NPB-baseline/LU/init_comm.f NPB-mod0/LU/init_comm.f
31c31
<       call MPI_INIT( IERROR )
---
> c      call MPI_INIT( IERROR )
diff -r NPB-baseline/LU/lu.f NPB-mod0/LU/lu.f
47c47,48
<       program applu
---
>       subroutine applu
> c      program applu
Only in NPB-mod0/LU: lu_wrapper.c
Only in NPB-baseline: MG
Only in NPB-baseline: MPI_dummy
diff -r NPB-baseline/Makefile NPB-mod0/Makefile
9,15c9,15
< BT: bt
< bt: header
<       cd BT; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
<
< SP: sp
< sp: header
<       cd SP; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
---
> #BT: bt
> #bt: header
> #     cd BT; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
>
> #SP: sp
> #sp: header
> #     cd SP; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
21,39c21,39
< MG: mg
< mg: header
<       cd MG; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
<
< FT: ft
< ft: header
<       cd FT; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
<
< IS: is
< is: header
<       cd IS; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
<
< CG: cg
< cg: header
<       cd CG; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
<
< EP: ep
< ep: header
<       cd EP; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
---
> #MG: mg
> #mg: header
> #     cd MG; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
>
> #FT: ft
> #ft: header
> #     cd FT; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
>
> #IS: is
> #is: header
> #     cd IS; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
>
> #CG: cg
> #cg: header
> #     cd CG; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
>
> #EP: ep
> #ep: header
> #     cd EP; $(MAKE) NPROCS=$(NPROCS) CLASS=$(CLASS)
59,60c59,60
<       - rm bin/sp.* bin/lu.* bin/mg.* bin/ft.* bin/bt.* bin/is.* bin/ep.* bin/cg.*
<
---
> #     - rm bin/sp.* bin/lu.* bin/mg.* bin/ft.* bin/bt.* bin/is.* bin/ep.* bin/cg.*
>       - rm  bin/lu.*
Only in NPB-baseline: SP
Only in NPB-baseline/config: NAS.samples
diff -r NPB-baseline/config/make.def NPB-mod0/config/make.def
40c40
< FMPI_LIB  = -L/usr/mpich/lib/LINUX/ch_p4 -lmpi -lfmpi
---
> FMPI_LIB  = -L/usr/mpich/lib/LINUX/ch_p4 -lmpi
50c50
< FFLAGS       = -O
---
> FFLAGS       = -fno-second-underscore -O
Only in NPB-baseline/config: make.def.template
Only in NPB-baseline/config: make.dummy
diff -r NPB-baseline/config/suite.def NPB-mod0/config/suite.def
14,22c14,29
< # The following example builds 1 processor sample sizes of all benchmarks.
< ft    S      1
< mg    S      1
< sp    S      1
< lu    S      1
< bt    S      1
< is    S      1
< ep    S      1
< cg    S      1
---
> lu    W      1
> lu    W      2
> lu    W      4
> lu    W      8
> lu    A      1
> lu    A      2
> lu    A      4
> lu    A      8
> lu    B       1
> lu    B       2
> lu    B       4
> lu    B       8
> lu    C       1
> lu    C       2
> lu    C       4
> lu    C       8
Only in NPB-baseline/config: suite.def.template
```

B.2. diff -r NPB-mod0 NPB-mod1a

```
diff -r NPB-mod0/LU/applu.incl NPB-mod1a/LU/applu.incl
6a7,21
> c
> c     MODIFICATIONS
> c     5 Dec 98 -- Changed declaration of the buf & buf1 arrays
> c                 to accomodate columnwise block striping
> c                 -- Necessary since the original declarations were
> c                    for (5,2*isiz2*isiz3), but this was under the
> c                    assumption (design) that isiz2.ge.isiz1
> c                    -- This was okay for original block-checkerboard
> c                       partitioning
> c                    -- This was okay for rowwise block striping
> c                    -- This is not a valid assumption for columnwise
> c                       block striping
> c
> c------------------------------------------------------------------
> c------------------------------------------------------------------
140,141c155,160
<       double precision  buf(5,2*isiz2*isiz3),
<      >                  buf1(5,2*isiz2*isiz3)
---
> c <MODIFICATIONS (mod1a)>
> c      double precision  buf(5,2*isiz2*isiz3),
> c     >                  buf1(5,2*isiz2*isiz3)
>       double precision  buf(5,2*isiz1*isiz3),
>      >                  buf1(5,2*isiz1*isiz3)
> c </MODIFICATIONS (mod1a)>
diff -r NPB-mod0/LU/exchange_4.f NPB-mod1a/LU/exchange_4.f
8a9,23
> c
> c     MODIFICATIONS
> c     8 Dec 98 -- Changed declaration of the g & h arrays
> c                 to accomodate columnwise block striping
> c                 -- Necessary since the original declarations were
> c                    for (0:isiz2+1,0:isiz3+1), but this was under the
> c                    assumption (design) that isiz2.ge.isiz1
> c                    -- This was okay for original block-checkerboard
> c                       partitioning
> c                    -- This was okay for rowwise block striping
> c                    -- This is not a valid assumption for columnwise
> c                       block striping
> c
> c------------------------------------------------------------------
> c------------------------------------------------------------------
22,23c37,42
<       double precision  g(0:isiz2+1,0:isiz3+1),
<      >        h(0:isiz2+1,0:isiz3+1)
---
> c <MODIFICATIONS (mod1a)>
> c      double precision  g(0:isiz2+1,0:isiz3+1),
> c     >        h(0:isiz2+1,0:isiz3+1)
>       double precision  g(0:isiz1+1,0:isiz3+1),
>      >        h(0:isiz1+1,0:isiz3+1)
> c </MODIFICATIONS (mod1a)>
diff -r NPB-mod0/LU/exchange_5.f NPB-mod1a/LU/exchange_5.f
8a9,27
> c
> c     MODIFICATIONS
> c     8 Dec 98 -- Changed declaration of the g array
> c                 to accomodate columnwise block striping
> c                 -- Necessary since the original declarations were
> c                    for (0:isiz2+1,0:isiz3+1), but this was under the
> c                    assumption (design) that isiz2.ge.isiz1
> c                    -- This was okay for original block-checkerboard
> c                       partitioning
> c                    -- This was okay for rowwise block striping
> c                    -- This is not a valid assumption for columnwise
> c                       block striping
> c                 -- Actually, I doubt this will have any impact, since
> c                    this is north-south comm, and with columnwise block
> c                     striping, there is but one row, but it is necessary
> c                     to provide a good interface with pintgr()
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
22c41,44
<       double precision  g(0:isiz2+1,0:isiz3+1)
---
> c <MODIFICATIONS (mod1a)>
> c      double precision  g(0:isiz2+1,0:isiz3+1)
>       double precision  g(0:isiz1+1,0:isiz3+1)
> c </MODIFICATIONS (mod1a)>
diff -r NPB-mod0/LU/exchange_6.f NPB-mod1a/LU/exchange_6.f
8a9,23
> c
> c     MODIFICATIONS
> c     8 Dec 98 -- Changed declaration of the g array
> c                 to accomodate columnwise block striping
> c                 -- Necessary since the original declarations were
> c                    for (0:isiz2+1,0:isiz3+1), but this was under the
> c                    assumption (design) that isiz2.ge.isiz1
> c                    -- This was okay for original block-checkerboard
> c                       partitioning
> c                    -- This was okay for rowwise block striping
> c                    -- This is not a valid assumption for columnwise
> c                       block striping
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
22c37,40
<       double precision  g(0:isiz2+1,0:isiz3+1)
---
> c <MODIFICATIONS (mod1a)>
> c      double precision  g(0:isiz2+1,0:isiz3+1)
>       double precision  g(0:isiz1+1,0:isiz3+1)
> c </MODIFICATIONS (mod1a)>
diff -r NPB-mod0/LU/pintgr.f NPB-mod1a/LU/pintgr.f
8a9,23
> c
> c     MODIFICATIONS
> c     8 Dec 98 -- Changed declaration of the phi1 & phi2 arrays
> c                 to accomodate columnwise block striping
> c                 -- Necessary since the original declarations were
> c                    for (0:isiz2+1,0:isiz3+1), but this was under the
> c                    assumption (design) that isiz2.ge.isiz1
> c                    -- This was okay for original block-checkerboard
> c                       partitioning
> c                    -- This was okay for rowwise block striping
> c                    -- This is not a valid assumption for columnwise
> c                       block striping
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
24,25c39,44
<       double precision  phi1(0:isiz2+1,0:isiz3+1),
<      >                  phi2(0:isiz2+1,0:isiz3+1)
---
> c <MODIFICATIONS (mod1a)>
> c      double precision  phi1(0:isiz2+1,0:isiz3+1),
> c     >                  phi2(0:isiz2+1,0:isiz3+1)
>       double precision  phi1(0:isiz1+1,0:isiz3+1),
>      >                  phi2(0:isiz1+1,0:isiz3+1)
> c </MODIFICATIONS (mod1a)>
59c78,81
<       do i = 0,isiz2+1
---
> c <MODIFICATIONS>
> c      do i = 0,isiz2+1
>       do i = 0,isiz1+1
> c </MODIFICATIONS>
126c148,151
<       do i = 0,isiz2+1
```

```
---
> c <MODIFICATIONS>
> c       do i = 0,isiz2+1
>       do i = 0,isiz1+1
> c </MODIFICATIONS>
205c230,233
<       do i = 0,isiz2+1
---
> c <MODIFICATIONS>
> c       do i = 0,isiz2+1
>       do i = 0,isiz1+1
> c </MODIFICATIONS>
diff -r NPB-mod0/LU/proc_grid.f NPB-mod1a/LU/proc_grid.f
8a9,27
> c
> c     MODIFICATIONS
> c     2 Dec 98 -- Changed block-checkerboard partitioning to
> c                 rowwise block striping
> c                 -- This may affect performance
> c                 -- This will require fewer changes than would be
> c                    demanded if I were to keep block-checkerboard and
> c                    tried to work with different-sized blocks
> c                    -- All this code assumes at most one neighbor node in
> c                       each direction -- I can continue to assure that with
> c                       block-striping, so I don't have to modify the code
> c                       to accept multiple neighbors
> c     4 Dec 98 -- Changed rowwise block striping to
> c                 columnwise block striping
> c                 -- This should improve performance over rowwise block
> c                    striping (locality)
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
20,21c39,44
< c   set up a two-d grid for processors: column-major ordering of unknowns
< c   NOTE: assumes a power-of-two number of processors
---
> cXXXset up a two-d grid for processors: column-major ordering of unknownsXXXX
> cXXXNOTE: assumes a power-of-two number of processorsXXXXXXXXXXXXXXXXXXXXXXXX
> c
> c     set up a one-d grid (a row, if you will) for processors
> c     NOTE: no longer assumes a power-of-two number of processors, but
> c     I'm not going to change this official requirement
25,27c48,54
<       xdim   = 2**(ndim/2)
<       if (mod(ndim,2).eq.1) xdim = xdim + xdim
<       ydim   = num/xdim
---
> c <MODIFICATIONS (mod1a)>
> c      xdim   = 2**(ndim/2)
> c      if (mod(ndim,2).eq.1) xdim = xdim + xdim
> c      ydim   = num/xdim
> c
> c      row    = mod(id,xdim) + 1
> c      col    = id/xdim + 1
29,30c56,60
<       row    = mod(id,xdim) + 1
<       col    = id/xdim + 1
---
>       xdim = 1
>       ydim = num
>       row  = 1
>       col  = id + 1
> c </MODIFICATIONS (mod1a)>
diff -r NPB-mod0/sys/setparams.c NPB-mod1a/sys/setparams.c
7c7,13
<  * the number of nodes and class for which a benchmark is being built.
---
>  * the number of nodes and class for which a benchmark is being built.
>
>  * MODIFICATIONS
>  * 2 Dec 98 -- Changed write_lu_info() to accomodate the change from
>  *              block-checkerboard partitioning to rowwise block striping
>  * 4 Dec 98 -- Changed write_lu_info() to accomodate the change from
>  *              rowwise block striping to columnwise block striping
448a455
> /* MODIFIED 2 Dec 98 by cb */
466,468c473,478
<   xdiv = ydiv = ilog2(nprocs)/2;
<   if (xdiv+ydiv != ilog2(nprocs)) xdiv += 1;
<   xdiv = ipow2(xdiv); ydiv = ipow2(ydiv);
---
> /* <MODIFICATIONS (mod1a)> */
> /*   xdiv = ydiv = ilog2(nprocs)/2; */
> /*   if (xdiv+ydiv != ilog2(nprocs)) xdiv += 1; */
> /*   xdiv = ipow2(xdiv); ydiv = ipow2(ydiv); */
>   xdiv = 1; ydiv = nprocs;
> /* </MODIFICATIONS (mod1a)> */
```

B.3. diff -r NPB-mod1a NPB-mod4.2

```
diff -r NPB-mod1a/LU/Makefile NPB-mod4.2/LU/Makefile
7,11c7,12
< OBJS = lu_wrapper.o lu.o init_comm.o read_input.o bcast_inputs.o proc_grid.o neighbors.o
\
<        nodedim.o subdomain.o setcoeff.o sethyper.o setbv.o exact.o setiv.o \
<        erhs.o ssor.o exchange_1.o exchange_3.o exchange_4.o exchange_5.o \
<        exchange_6.o rhs.o l2norm.o jacld.o blts.o jacu.o buts.o error.o \
<        pintgr.o verify.o ${COMMON}/print_results.o ${COMMON}/timers.o
---
> OBJS = lu_wrapper.o lu.o init_comm.o read_input.o bcast_inputs.o proc_grid.o \
>        neighbors.o nodedim.o subdomain.o setcoeff.o sethyper.o setbv.o \
>        exact.o setiv.o erhs.o ssor.o exchange_1.o exchange_3.o exchange_4.o \
>        exchange_5.o exchange_6.o rhs.o l2norm.o jacld.o blts.o jacu.o buts.o \
>        error.o pintgr.o verify.o ${COMMON}/print_results.o ${COMMON}/timers.o \
>        get_name.o weighnode.o metric.o metricmap.o
31a33,35
> get_name.o:  get_name.c
>       ${CCOMPILE} get_name.c
>
62c66,67
< subdomain.o: subdomain.f applu.incl npbparams.h mpinpb.h
---
> subdomain.o: subdomain.f applu.incl npbparams.h mpinpb.h \
>              ../metric/weighnode.h ../metric/metric.h ../metric/metricmap.h
diff -r NPB-mod1a/LU/blts.f NPB-mod4.2/LU/blts.f
13a14,20
> c
> c     MODIFICATIONS
> c     21 Dec 98 -- Added "upshot" instrumentation
> c     22 Dec 98 -- Thinned out "upshot" instrumentation
> c
> c---------------------------------------------------------------------
> c---------------------------------------------------------------------
54a62,64
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(12,0,"Start Exchange")
> c </MODIFICATIONS (mod2.x)>
55a66,68
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(13,0,"End Exchange")
> c </MODIFICATIONS (mod2.x)>
255a269,271
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(12,0,"Start Exchange")
> c </MODIFICATIONS (mod2.x)>
256a273,275
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(13,0,"End Exchange")
> c </MODIFICATIONS (mod2.x)>
diff -r NPB-mod1a/LU/buts.f NPB-mod4.2/LU/buts.f
13a14,20
> c
> c     MODIFICATIONS
> c     21 Dec 98 -- Added "upshot" instrumentation
> c     22 Dec 98 -- Thinned out "upshot" instrumentation
> c
> c---------------------------------------------------------------------
> c---------------------------------------------------------------------
55a63,65
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(12,0,"Start Exchange")
> c </MODIFICATIONS (mod2.x)>
56a67,69
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(13,0,"End Exchange")
> c </MODIFICATIONS (mod2.x)>
255a269,271
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(12,0,"Start Exchange")
> c </MODIFICATIONS (mod2.x)>
```

```
256a273,275
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(13,0,"End Exchange")
> c </MODIFICATIONS (mod2.x)>
Only in NPB-mod4.2/LU: get_name.c
diff -r NPB-mod1a/LU/rhs.f NPB-mod4.2/LU/rhs.f
8a9,15
> c
> c     MODIFICATIONS
> c     21 Dec 98 -- Added "upshot" instrumentation
> c     22 Dec 98 -- Thinned out "upshot" instrumentation
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
60a68,70
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(12,0,"Start Exchange")
> c </MODIFICATIONS (mod2.x)>
61a72,74
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(13,0,"End Exchange")
> c </MODIFICATIONS (mod2.x)>
215a229,231
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(12,0,"Start Exchange")
> c </MODIFICATIONS (mod2.x)>
216a233,235
> c <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(13,0,"End Exchange")
> c </MODIFICATIONS (mod2.x)>
diff -r NPB-mod1a/LU/ssor.f NPB-mod4.2/LU/ssor.f
7a8,20
> c
> c     MODIFICATIONS
> c     11 Dec 98 -- Added some instrumentation to display underlying
> c                  information about each node
> c     21 Dec 98 -- Refined instrumentation to display underlying
> c                  information about each node
> c               -- Added "upshot" instrumentation
> c     22 Dec 98 -- Thinned out "upshot" instrumentation
> c     23 Dec 98 -- Moved "print" instrumentation to subdomain()
> c      4 Jan 99 -- Removed instrumentation
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
29a43,48
> c <MODIFICATIONS (mod3.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      character*32 p_name
> c      integer n_len
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod3.x)>
59a79,105
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_INIT_LOG
> c      if (id.eq.0) then
> cc          call MPE_DESCRIBE_STATE(1,2,"SSOR","2x2")
> cc          call MPE_DESCRIBE_STATE(3,4,"Lower Triangle","dllines3")
> cc          call MPE_DESCRIBE_STATE(5,6,"Upper Triangle","drlines3")
> cc          call MPE_DESCRIBE_STATE(8,9,"L2-Norm","dimple3")
> cc          call MPE_DESCRIBE_STATE(10,11,"SS Residuals","vlines3")
> cc          call MPE_DESCRIBE_STATE(12,13,"Exchange","black")
> cc          call MPE_DESCRIBE_STATE(14,15,"Synchronize","boxes")
> cc
> cc          call MPE_DESCRIBE_STATE(1,2,"SSOR","blue")
> cc          call MPE_DESCRIBE_STATE(3,4,"Lower Triangle","red")
> cc          call MPE_DESCRIBE_STATE(5,6,"Upper Triangle","green")
> cc          call MPE_DESCRIBE_STATE(8,9,"L2-Norm","yellow")
> cc          call MPE_DESCRIBE_STATE(10,11,"SS Residuals","pink")
> cc          call MPE_DESCRIBE_STATE(12,13,"Exchange","purple")
> cc          call MPE_DESCRIBE_STATE(14,15,"Synchronize","orange")
> cc
> c         call MPE_DESCRIBE_STATE(3,4,"Lower Triangle","black")
> c         call MPE_DESCRIBE_STATE(5,6,"Upper Triangle","white")
> c         call MPE_DESCRIBE_STATE(10,11,"SS Residuals","gray")
> c      endif
> cc      call MPE_LOG_EVENT(10,0,"Start RHS")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
60a107,111
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc      call MPE_LOG_EVENT(11,0,"End RHS")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
64a116,120
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc      call MPE_LOG_EVENT(8,0,"Start L2Norm")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
67a124,128
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc      call MPE_LOG_EVENT(9,0,"End L2Norm")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
76c137,142
<
---
>
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(14,0,"Start Barrier")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
77a144,148
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(15,0,"End Barrier")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
83a155,159
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc      call MPE_LOG_EVENT(1,0,"Start SSOR")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
115a192,196
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(3,0,"Start LT")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
133c214,224
<
---
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(4,0,"End LT")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
>
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(5,0,"Start UT")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
150a242,246
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c      call MPE_LOG_EVENT(6,0,"End UT")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
170a267,271
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc       call MPE_LOG_EVENT(8,0,"Start L2Norm")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
173a275,279
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc       call MPE_LOG_EVENT(9,0,"End L2Norm")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
183a290,294
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c       call MPE_LOG_EVENT(10,0,"Start RHS")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
184a296,300
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> c       call MPE_LOG_EVENT(11,0,"End RHS")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
190a307,311
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc       call MPE_LOG_EVENT(8,0,"Start L2Norm")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
193a315,319
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc       call MPE_LOG_EVENT(9,0,"End L2Norm")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
214a341,345
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod2.x)>
> cc       call MPE_LOG_EVENT(2,0,"End SSOR")
> cc </MODIFICATIONS (mod2.x)>
> c </MODIFICATIONS (mod4.x)>
217a349,370
> c <MODIFICATIONS (mod4.x)>
> cc <MODIFICATIONS (mod3.x)>
> ccc <MODIFICATIONS (mod2.x)>
> c       call MPE_FINISH_LOG("LU")
> c
> cc       call get_name(p_name,n_len)
> cc       print 2700,id,p_name(1:n_len)
> cc       print 2701,id,nx,ny,nz
> cc       print 2704,id,ipt,ipt+nx-1,jpt,jpt+ny-1,0,nz-1
> cc       print 2702,id,north,south,west,east
> ccc       print 2703,id,ist,iend,jst,jend
> cc 2700 format ('Process',i3,' executing on ',A)
> cc 2701 format ('Process',i3,' dimensions:  nx=',i3,' ny=',i3,' nz=',i3)
> cc 2702 format ('Process',i3,' neighbors:  north=',i3,' south=',i3,
> cc     >      ' west=',i3,' east=',i3)
> ccc 2703 format ('Process',i3,' position:  ist=',i3,' iend=',i3,
> ccc     >      ' jst=',i3,' jend=',i3)
> cc 2704 format ('Process',i3,' position:  ipt=',i3,' ..',i3,
> cc     >      ' jpt=',i3,' ..',i3,' kpt=',i3,' ..',i3)
> ccc </MODIFICATIONS (mod2.x)>
> cc </MODIFICATIONS (mod3.x)>
> c </MODIFICATIONS (mod4.x)>
diff -r NPB-mod1a/LU/subdomain.f NPB-mod4.2/LU/subdomain.f
8a9,30
> c
> c     MODIFICATIONS
> c     23 Dec 98 -- Moved "print" instrumentation from ssor()
> c               -- Interfaced with weighnode()
> c              -- Adjusted partitioning scheme, instrumenting as I go
> c     24 Dec 98 -- Fleshed-out the partition refinement code (yesterday,
> c                 I left it as stubs to abort if the sum of the subdomains
> c                 was not equal to the domain itself, or if any partition
> c                 was smaller than four rows/columns thick)
> c                 -- Two ways to refine partition:
> c                    -- Steal from poor, give to rich
> c                       -- If there's extra work to be done, give it to the
> c                          strong-muscly-types; if there's too much work,
> c                          give the 90-pound-weaklings a break
> c                    -- Steal from rich, give to poor
> c                       -- Avoid overcompensating
> c                       -- Won't induce partition-too-small problem like
> c                          the first option would
> c      4 Jan 99 -- Removed instrumentation
> c
> c-------------------------------------------------------------------
> c-------------------------------------------------------------------
17a40,54
> c <MODIFICATIONS (mod3.x)>
> c Function declaration
>       double precision weighnode
> c Variable declaration
>       character*32 p_name
>       integer n_len,loop,sum,itemp
>       integer iarg1,iarg2,iarg3,iarg4
>       double precision weight,farg2,temp
>       double precision glblw8(0:nnodes_compiled-1),ttlw8
> c              get it?  glblw8 --> global w8 --> global weight
> c              boy, I'm too funny!
>       integer isiz0t,nt(0:nnodes_compiled-1),tpt(0:nnodes_compiled-1)
>       logical sorted
>       integer lo_end,hi_end,pointer(0:nnodes_compiled-1)
> c </MODIFICATIONS (mod3.x)>
26a64,66
> c <MODIFICATIONS (mod3.x)>
> c Leave the original calculations there for compairson purposes
> c </MODIFICATIONS (mod3.x)>
55a96,232
> c <MODIFICATIONS (mod3.x)>
> c <MODIFICATIONS (mod4.x)>
> c      call get_name(p_name,n_len)
> c      print 3600,id,p_name(1:n_len)
> c      print 3601,id,north,south,west,east
> c      print 3602,id,nx,ny,nz
> c      print 3603,id,ipt,ipt+nx-1,jpt,jpt+ny-1,0,nz-1
> c 3600 format ('Process',i3,' executing on ',A)
> c 3601 format ('Process',i3,' original neighbors:  north=',i3,' south='
> c     >      ,i3,' west=',i3,' east=',i3)
> c 3602 format ('Process',i3,' original dimensions:  nx=',i3,' ny=',i3,
> c     >      ' nz=',i3)
> c 3603 format ('Process',i3,' original position:  ipt=',i3,' ..',i3,
> c     >      ' jpt=',i3,' ..',i3,' kpt=',i3,' ..',i3)
> c </MODIFICATIONS (mod4.x)>
>
> c-------------------------------------------------------------------
> c   weigh each node and partition grid appropriately
> c-------------------------------------------------------------------
>       isiz0t = isiz01
>
> c      iarg1 = 0                                 ! Programmer-specified
> c      farg2 = 1.0                               ! Programmer-specified
> c      weight = weighnode(iarg1,farg2)           ! Programmer-specified
> c      iarg1 = 1                                 ! /proc/cpuinfo
> c      iarg2 = 0                                 ! /proc/cpuinfo
> c      weight = weighnode(iarg1,iarg2)           ! /proc/cpuinfo
> c      iarg1 = 1                                 ! /proc/cpuinfo converted
> c      iarg2 = 1                                 ! /proc/cpuinfo converted
> c      iarg3 = 4                                 ! /proc/cpuinfo converted
> c      weight = weighnode(iarg1,iarg2,iarg3)     ! /proc/cpuinfo converted
>       iarg1 = 2                                 ! calc_pi
>       iarg2 = 22                                ! calc_pi
>       iarg3 = 0                                  ! calc_pi
>       weight = weighnode(iarg1,iarg2,iarg3)      ! calc_pi
> c      iarg1 = 2                                  ! calc_pi converted
> c      iarg2 = 22                                 ! calc_pi converted
> c      iarg3 = 1                                  ! calc_pi converted
> c      iarg4 = 4                                  ! calc_pi converted
> c      weight = weighnode(iarg1,iarg2,iarg3,iarg4) ! calc_pi converted
> c <MODIFICATIONS (mod4.x)>
> c      print 3604,id,weight,p_name(1:n_len)
> c 3604 format ('Process',i3,' reports weight=',f14.2,
> c     >      ' while executing on ',A)
> c </MODIFICATIONS (mod4.x)>
>
>
>       call MPI_ALLGATHER(weight,1,MPI_DOUBLE_PRECISION,
>      >     glblw8,1,MPI_DOUBLE_PRECISION,
>      >     MPI_COMM_WORLD, IERROR)
>
>       ttlw8 = 0.0
>       do 3651 loop=0,nnodes_compiled-1
>          ttlw8 = ttlw8 + glblw8(loop)
>  3651 continue
>       sum = 0
>       do 3652 loop=0,nnodes_compiled-1
>          temp = glblw8(loop)*isiz0t               ! common subexpression
>          nt(loop) = temp/ttlw8                    ! nt is int, so truncated
>          if (mod(temp,ttlw8)/ttlw8.ge.0.5) then
>             nt(loop) = nt(loop)+1                 ! correct rounding error
>          endif
>          sum = sum+nt(loop)                       ! to check the math later
>          pointer(loop) = loop                     ! initialize pointers
>  3652 continue
>
> c bubblesort may not be the most scalable sort in the world, but it's
> c quick'n'easy to code, and we're not exactly dealing with a large number
> c of processors here -- the overhead of something like quicksort may be even
> c worse for our small number of processors
> c to give credit where it's due, this is from D.M. Etter, /Structured
> c  Fortran 77 for Engineers and Scientists/.  Menlo Park CA:  The
> c Benjamin/Cummings Publishing Company, 1987, p193, with some modifications
> c (variable names (big deal) and the use of pseudopointers)
>       sorted = .false.
>  3655 if (.not.sorted) then
>          sorted = .true.
>          do 3656 loop=0,nnodes_compiled-2
>             if (nt(pointer(loop)).gt.nt(pointer(loop+1))) then
>                itemp = pointer(loop)
>                pointer(loop) = pointer(loop+1)
>                pointer(loop+1) = itemp
>                sorted = .false.
>             endif
>  3656    continue
>          go to 3655
>       endif
>       lo_end = 0                                  ! steal from the poor
>       hi_end = nnodes_compiled-1                  ! give to the rich
>
>       if (sum.ne.isiz0t) then                     ! nuts
>  3657    if (sum.gt.isiz0t) then                  ! ease the lowend's load
>             nt(pointer(lo_end)) = nt(pointer(lo_end))-1
>             lo_end = lo_end+1                     ! share the easement
>             sum = sum-1
>             go to 3657                            ! make sure we're finished
>          endif
>  3658    if (sum.lt.isiz0t) then                  ! more work for highend
>             nt(pointer(hi_end)) = nt(pointer(hi_end))+1
>             hi_end = hi_end-1                     ! share the extra effort
>             sum = sum+1
>             go to 3658                            ! make sure we're done
>          endif
>       endif
>
```

```
>       do 3659 loop=0,nnodes_compiled-2
>          if (nt(pointer(loop)).lt.4) then          ! nuts
>             itemp = 4-nt(pointer(loop))
>             nt(pointer(loop)) = nt(pointer(loop))+itemp
>             nt(pointer(loop+1)) = nt(pointer(loop+1))-itemp
>          endif
>  3659 continue
>       if (nt(pointer(nnodes_compiled-1)).lt.4) then ! gosh darn it
>       endif                        ! do nothing ... it'll get caught below
>
>       tpt(0) = 0
>       do 3654 loop=1,nnodes_compiled-1
>          tpt(loop) = tpt(loop-1)+nt(loop-1)
>  3654 continue
>
> c </MODIFICATIONS (mod3.x)>
> c <MODIFICATIONS (mod3.2)>
>       ny = nt(id)
>       jpt = tpt(id)
>
> c </MODIFICATIONS (mod3.2)>
> c <MODIFICATIONS (mod4.x)>
> c <MODIFICATIONS (mod3.x)>
> c      print 3613,id,nx,ny,nz
> c      print 3614,id,ipt,ipt+nx-1,jpt,jpt+ny-1,0,nz-1
> c 3613 format ('Process',i3,' new dimensions:  nx=',i3,' ny=',i3,
> c     >      ' nz=',i3)
> c 3614 format ('Process',i3,' new position:  ipt=',i3,' ..',i3,
> c     >      ' jpt=',i3,' ..',i3,' kpt=',i3,' ..',i3)
> c
> c </MODIFICATIONS (mod3.x)>
> c </MODIFICATIONS (mod4.x)>
diff -r NPB-mod1a/Makefile NPB-mod4.2/Makefile
18a19,20
>       cd metric; make all; make HINT BINDIR=../bin
>       cp metric/*.o LU;
55a58
>       - rm -rf bin/hint
60a64
>       - rm -f metric/hint/core metric/hint/*.o metric/hint/*~
diff -r NPB-mod1a/config/make.def NPB-mod4.2/config/make.def
40c40
< FMPI_LIB  = -L/usr/mpich/lib/LINUX/ch_p4 -lmpi
---
> FMPI_LIB  = -L/usr/mpich/lib/LINUX/ch_p4 -lmpe -lpmpi -lmpi
87c87
< CMPI_LIB  = -L/usr/mpich/lib/LINUX/ch_p4 -lmpi
---
> CMPI_LIB  = -L/usr/mpich/lib/LINUX/ch_p4 -lmpe -lpmpi -lmpi
Only in NPB-mod4.2: metric
diff -r NPB-mod1a/sys/setparams.c NPB-mod4.2/sys/setparams.c
13a14,18
>  * 24 Dec 98 -- Changed write_lu_info() to allocate the entire problem
>  *              size for each processor -- I know this is a waste of (virtual)
>  *              memory, but without dynamic memory allocation, it's the only
>  *              way I can be assured I'll have enough memory for a non-
>  *              fixed partitioning
455a461,462
> /* MODIFIED 4 Dec 98 by cb */
> /* MODIFIED 24 Dec 98 by cb */
479,480c486,491
<   isiz1 = problem_size/xdiv; if (isiz1*xdiv < problem_size) isiz1++;
<   isiz2 = problem_size/ydiv; if (isiz2*ydiv < problem_size) isiz2++;
---
> /* <MODIFICATIONS (mod3.x)> */
> /*   isiz1 = problem_size/xdiv; if (isiz1*xdiv < problem_size) isiz1++; */
> /*   isiz2 = problem_size/ydiv; if (isiz2*ydiv < problem_size) isiz2++; */
>   isiz1 = problem_size;
>   isiz2 = problem_size;
> /* </MODIFICATIONS (mod3.x)> */
```

B.4. diff -r NPB-mod4.2 NPB-mod4.3

```
diff -r NPB-mod4.2/LU/proc_grid.f NPB-mod4.3/LU/proc_grid.f
45a46,48
> c     22 Jan 99 -- yes, I am.  I want to run this thing on all processors
> c                  on the system, and block striping allows me to do that
> c
diff -r NPB-mod4.2/config/suite.def NPB-mod4.3/config/suite.def
14,29c14,22
< lu    W       1
< lu    W       2
< lu    W       4
< lu    W       8
< lu    A       1
< lu    A       2
< lu    A       4
< lu    A       8
< lu    B       1
< lu    B       2
< lu    B       4
< lu    B       8
< lu    C       1
< lu    C       2
< lu    C       4
< lu    C       8
---
> lu    A       10
> lu    A       11
> lu    A       12
> lu    B       10
> lu    B       11
> lu    B       12
> lu    C       10
> lu    C       11
> lu    C       12
diff -r NPB-mod4.2/sys/setparams.c NPB-mod4.3/sys/setparams.c
18a19
>  * 22 Jan 99 -- Removed power-of-two requirement from LU
218c219,221
<   case LU:
---
> /* <MODIFICATIONS (x.3)> */
> /*   case LU: */
> /* </MODIFICATIONS (x.3)> */
226a230,232
> /* <MODIFICATIONS (x.3)> */
>   case LU:
> /* </MODIFICATIONS (x.3)> */
```

B.5. MPI Wrappers

B.5.1. lu_wrapper.c

```
#include "mpi.h"

extern void applu_();

int main(int argc, char *argv[]) {
  MPI_Init(&argc,&argv);
  applu_();
  return 0;
}
```

B.5.2. get_name.c

```
#include "mpi.h"

void get_name_(char *procname,int *namelen) {
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Get_processor_name(procname,namelen);
}
```

Appendix C: NodeMetric Source Code

This source code is stored on the ABC in the `/home/cbohn/thesis` directory, and it is also available directly from the author (see Vita for contact information).

C.1. weighnode

C.1.1. ***weighnode.h***

```
/**********************************************************************
 *                                                                    *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous              *
 *          Cluster of PCs                                            *
 *          AFIT/GE/ENG/99M-02                                        *
 *                                                                    *
 * PACKAGE: NodeMetric                                                *
 *          0.2                                                       *
 *                                                                    *
 * FILE:    weighnode.h                                               *
 *          0.1                                                       *
 *          Single interface for a program to assess the relative     *
 *          processing power (computational & other) of a compute     *
 *          node                                                      *
 *                                                                    *
 * AUTHOR:  Capt Christopher A. Bohn                                  *
 *                                                                    *
 * HISTORY: 26 Nov 98 -- Version a.1 begun                            *
 *          27 Nov 98 -- a.1 complete                                 *
 *                    -- Version a.2 begun                            *
 *                    -- a.2 complete                                 *
 *                    -- Version a.3 begun                            *
 *                    -- a.3 complete                                 *
 *          28 Nov 98 -- Version a.4 begun                            *
 *                    -- a.4 complete                                 *
 *                    -- Version a.5 begun                            *
 *          29 Nov 98 -- a.5 (C version) complete -- FORTRAN versions *
 *                       abandoned                                    *
 *                    -- Version a.5.1 begun                          *
 *                    -- a.5.1 (FORTRAN versions) complete            *
 *                    -- weighnode.h Version 0.1                      *
 *                    -- NodeMetric Version 0.1                       *
 *          11 Dec 98 -- NodeMetric 0.1.1                             *
 *          31 Dec 98 -- NodeMetric 0.1.2                             *
 *           1 Jan 99 -- NodeMetric 0.2                               *
 *                                                                    *
 **********************************************************************/


double weighnode (int*, ...);
double weighnode_ (int*, ...);
double weighnode__ (int*, ...);


/* weighnode */
  /*********************************************************************
```

```
Uses parameters (all pass-by-reference) to determine how a
compute node should be evaluated.  Interfaces with metric.c &
metricmap.c to accomplish the actual measurement.  Returns weight, or
returns 0.0 in case of error.
  Parameters are:
     Parameter 1 -- Metric
        0 -- Programmer-specified
             -- seemingly pointless option, but I can imagine a couple
                instances in which the application programmer might
                want this option
            Parameter 2 -- weight to return to the application (double)
            Parameters 3 & 4 -- ignored (optional)
        1 -- Parse /proc/cpuinfo
             -- only works on Linux systems
            -- current implementation only parses in bogomips
            -- current implementation assumes uniprocessor
            Parameter 2 -- Return original weight or attempt to unskew?
                 0 -- original weight
                 1 -- convert weight
             Parameter 3 -- ignored (optional) if original weight is
                              desired
                           -- if weight is to be converted, then
                 0 -- convert for 16-bit integer operations
                 1 -- convert for 32-bit integer operations
                 2 -- convert for 64-bit integer operations
                 3 -- convert for 32-bit floating point operations
                 4 -- convert for 64-bit floating point operations
            Parameter 4 -- ignored (optional)
         2 -- Determine Mflops performance by calculating Pi
            Parameter 2 -- indicate level of precision
                        -- current valid values are 16-28
                          -- less than 16 and time frame is too small
                             to measure
                         -- more than 28 and the operations count
                             overflows
                       -- personally, I recommend 22, maybe 23 to get
                          the steady-state Mflops reading in minimal
                         time
            Parameter 3 -- Return original weight or attempt to unskew?
                 0 -- original weight
                 1 -- convert weight
             Parameter 4 -- ignored (optional) if original weight is
                              desired
                           -- if weight is to be converted, then
                 0 -- convert for 16-bit integer operations
                 1 -- convert for 32-bit integer operations
                 2 -- convert for 64-bit integer operations
                 3 -- convert for 32-bit floating point operations
                 4 -- convert for 64-bit floating point operations
        3 -- Determine QUIPS performance by using HINT benchmark
             Parameter 2 -- Specify nature of operations
                 0 -- convert for 16-bit integer operations
                 1 -- convert for 32-bit integer operations
                 2 -- convert for 64-bit integer operations
                 3 -- convert for 32-bit floating point operations
                 4 -- convert for 64-bit floating point operations
            Parameters 3 & 4 -- ignored (optional)
********************************************************************/
```

C.1.2. ***weighnode.c***

```
#include <stdarg.h>
#include "metric.h"
#include "metricmap.h"

/**********************************************************************
 *                                                                    *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous             *
 *          Cluster of PCs                                            *
 *          AFIT/ENG/GE99M-02                                         *
 *                                                                    *
 * PACKAGE: NodeMetric                                                *
 *          0.2                                                       *
 *                                                                    *
 * FILE:    weighnode.c                                               *
 *          0.1                                                       *
 *          Single interface for a program to assess the relative    *
 *          processing power (computational & other) of a compute    *
 *          node                                                      *
 *                                                                    *
 * AUTHOR:  Capt Christopher A. Bohn                                  *
 *                                                                    *
 * HISTORY: 26 Nov 98 -- Version a.1 begun                            *
 *                       -- Objective: Develop interface for          *
 *                          weighnode, capable of being called from   *
 *                          FORTRAN transparently, as well            *
 *          27 Nov 98 -- a.1 complete                                 *
 *                    -- Version a.2 begun                            *
 *                       -- Objective: Correctly interpret variable   *
 *                          arguments (here's hoping that the man     *
 *                          page and about 1 printed page's worth of  *
 *                          a textbook (Kelley & Pohl, A Book on C,   *
 *                          Benjamin Cummings, 1990, pp462-463) is    *
 *                          sufficient)                               *
 *                    -- a.2 complete                                 *
 *                    -- Version a.3 begun                            *
 *                       -- Objective: Implement the FORTRAN versions *
 *                          (well, they're written in C, but they     *
 *                          have trailing underscores) that simply    *
 *                          call the C version                        *
 *                    -- a.3 complete                                 *
 *          28 Nov 98 -- Version a.4 begun                            *
 *                       -- Objective: Add capability for the         *
 *                          application programmer to specify the     *
 *                          weight that weighnode() will return --    *
 *                          I know this seems unnecessary, but a) I   *
 *                          have noticed that programmers tend to     *
 *                          find uses for features the original       *
 *                          programmer never though of;  b) I can     *
 *                          think of a couple uses for such an        *
 *                          option (suppose the app prog'r wants to   *
 *                          use a b/m I'm not providing, or wants to  *
 *                          specify identical weights for all nodes,  *
 *                          and (s)he doesn't want to muck with the   *
 *                          code ... now the app prog'r only has to   *
 *                          change the weighnode() line ... no other  *
 *                          lines need be affected, and none of this  *
 *                          ugly commented-out line business; and     *
 *                          c) it's cheap ... these comments take up  *
 *                          more space in the source code than the    *
 *                          actual code will!                         *
 *                    -- a.4 complete                                 *
 *                    -- Version a.5 begun                            *
 *                       -- Objective: Let's code this puppy!         *
 *          29 Nov 98 -- a.5 (C version) complete -- FORTRAN versions *
 *                       abandoned, as I realized one of my           *
 *                       assumptions was not-so-good                  *
 *                       -- I had assumed the it would be trivial for *
 *                          weighnode_() & weighnode__() to call      *
 *                          weighnode(), and then I could keep code   *
 *                          maintenance simpler by only modifying     *
 *                           weighnode()                              *
 *                        -- Turns out the FORTRAN versions would     *
 *                           have to do a heckuva lot of decoding to  *
 *                           correctly call the C version (due to     *
 *                           variable arguments), and this decoding   *
 *                           would also have to be maintained         *
 *                     -- Version a.5.1 begun                         *
 *                        -- Objective: Copy the code from the C      *
 *                           version into the FORTRAN version         *
 *                     -- a.5.1 complete                              *
 *                     -- weighnode.c Version 0.1                     *
 *                     -- NodeMetric Version 0.1                      *
 *           11 Dec 98 -- NodeMetric 0.1.1                            *
 *           31 Dec 98 -- NodeMetric 0.1.2                            *
 *            1 Jan 99 -- NodeMetric 0.2                              *
 *                                                                    *
 **********************************************************************/


double weighnode (int *yardstick, ...) {
  /*********************************************************************
    Uses parameters (all pass-by-reference) to determine how a
    compute node should be evaluated.  Interfaces with metric.c &
    metricmap.c to accomplish the actual measurement.  Returns weight, or
    returns 0.0 in case of error.
      Parameters are:
         Parameter 1 -- Metric
            0 -- Programmer-specified
                 -- seemingly pointless option, but I can imagine a couple
                    instances in which the application programmer might
                    want this option
                 Parameter 2 -- weight to return to the application (double)
                 Parameters 3 & 4 -- ignored (optional)
            1 -- Parse /proc/cpuinfo
                 -- only works on Linux systems
                 -- current implementation only parses in bogomips
                 -- current implementation assumes uniprocessor
                 Parameter 2 -- Return original weight or attempt to unskew?
                      0 -- original weight
                      1 -- convert weight
                 Parameter 3 -- ignored (optional) if original weight is
                                desired
                             -- if weight is to be converted, then
                      0 -- convert for 16-bit integer operations
                      1 -- convert for 32-bit integer operations
                      2 -- convert for 64-bit integer operations
                      3 -- convert for 32-bit floating point operations
                      4 -- convert for 64-bit floating point operations
                 Parameter 4 -- ignored (optional)
             2 -- Determine Mflops performance by calculating Pi
                 Parameter 2 -- indicate level of precision
                             -- current valid values are 16-28
                                -- less than 16 and time frame is too small
                                   to measure
                                -- more than 28 and the operations count
                                   overflows
                            -- personally, I recommend 22, maybe 23 to get
                               the steady-state Mflops reading in minimal
                               time
                 Parameter 3 -- Return original weight or attempt to unskew?
                      0 -- original weight
                      1 -- convert weight
                 Parameter 4 -- ignored (optional) if original weight is
                                desired
                             -- if weight is to be converted, then
                      0 -- convert for 16-bit integer operations
                      1 -- convert for 32-bit integer operations
                      2 -- convert for 64-bit integer operations
                      3 -- convert for 32-bit floating point operations
                      4 -- convert for 64-bit floating point operations
            3 -- Determine QUIPS performance by using HINT benchmark
```

```
                Parameter 2 -- Specify nature of operations
                     0 -- convert for 16-bit integer operations
                     1 -- convert for 32-bit integer operations
                     2 -- convert for 64-bit integer operations
                     3 -- convert for 32-bit floating point operations
                     4 -- convert for 64-bit floating point operations
                Parameters 3 & 4 -- ignored (optional)
  26 Nov 98 -- Developed interface
  27 Nov 98 -- Finished interface
            -- "Mastered" variable arguments
            -- Implemented FORTRAN versions
  28 Nov 98 -- Added option to return programmer-specified weight
            -- Started implementation
  29 Nov 98 -- C version coded
            -- FORTRAN versions coded
*********************************************************************/
double weight,*weightp,pi;
int arg1,arg2,arg3,arg4;
int *arg2p,*arg3p,*arg4p;

va_list ap;
va_start(ap,yardstick);                  /* Initialize variable arguments */

arg1=*yardstick;
switch(arg1) {

case 0:                                  /* Let programmer specify weight */
  weightp=va_arg(ap,double*);            /* Return the second argument */
  weight=*weightp;
  break;

case 1:                                  /* Parse /proc/cpuinfo */
  arg2p=va_arg(ap,int*);
  arg2=*arg2p;
  if (arg2==0) weight=parse_cpuinfo();   /* Return bogomips, straight-up  */
  else {
    arg3p=va_arg(ap,int*);
    arg3=*arg3p;
    switch(arg3) {
    case 0:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"SHORT");
      break;                             /* Convert to HINT SHORT  */
    case 1:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"INT");
      break;                             /* Convert to HINT INT  */
    case 2:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"LONGLONG");
      break;                             /* Convert to HINT LONGLONG  */
    case 3:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"FLOAT");
      break;                             /* Convert to HINT FLOAT  */
    case 4:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"DOUBLE");
      break;                             /* Convert to HINT DOUBLE  */
    case 5:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"NAS");
      break;                             /* Convert to NAS NPB-W-serial  */
    default:
      weight=0.0;
    } /* switch (arg3) */
  } /* else (arg2!=0) */
  break;

case 2:                                  /* Calculate Pi & count flops */
  arg2p=va_arg(ap,int*);                 /* Level of precision */
  arg2=*arg2p;
  arg3p=va_arg(ap,int*);
  arg3=*arg3p;
  if (arg3==0) weight=calc_pi(pow(2,arg2),&pi);
                                         /* Return Mflops, straight-up  */
  else {
    arg4p=va_arg(ap,int*);
```

```
      arg4=*arg4p;
      switch(arg4) {
      case 0:
        weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"SHORT");
        break;                          /* Convert to HINT SHORT  */
      case 1:
        weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"INT");
        break;                          /* Convert to HINT INT  */
      case 2:
        weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"LONGLONG");
        break;                          /* Convert to HINT LONGLONG  */
      case 3:
        weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"FLOAT");
        break;                          /* Convert to HINT FLOAT  */
      case 4:
        weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"DOUBLE");
        break;                          /* Convert to HINT DOUBLE  */
      case 5:
        weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"NAS");
        break;                          /* Convert to NAS NPB-W-serial  */
      default:
        weight=0.0;
      } /* switch (arg4) */
    } /* else (arg3!=0) */
    break;

  case 3:                               /* Use HINT b/m to weigh nodes */
    arg2p=va_arg(ap,int*);
    arg2=*arg2p;
    switch(arg2) {
    case 0:
      weight=run_hint("SHORT");
      break;                            /* Convert to HINT SHORT  */
    case 1:
      weight=run_hint("INT");
      break;                            /* Convert to HINT INT  */
    case 2:
      weight=run_hint("LONGLONG");
      break;                            /* Convert to HINT LONGLONG  */
    case 3:
      weight=run_hint("FLOAT");
      break;                            /* Convert to HINT FLOAT  */
    case 4:
      weight=run_hint("DOUBLE");
      break;                            /* Convert to HINT DOUBLE  */
    default:
      weight=0.0;
    } /* switch (arg2) */
    break;

  case 4:                               /* Use NAS NPB-W-serial */
    weight=0.0;                         /* Not yet available */
    break;

  default:
    weight=0.0;
  } /* switch (arg1) */

  va_end(ap);                           /* Wrap things up */
  return weight;
} /* weighnode */


double weighnode_(int *yardstick, ...) {
  /*******************************************************************
    See header for weighnode() for description & history
  *******************************************************************/
  double weight,*weightp,pi;
  int arg1,arg2,arg3,arg4;
  int *arg2p,*arg3p,*arg4p;
```

```
va_list ap;
va_start(ap,yardstick);                    /* Initialize variable arguments */

arg1=*yardstick;
switch(arg1) {

case 0:                                    /* Let programmer specify weight */
  weightp=va_arg(ap,double*);              /* Return the second argument */
  weight=*weightp;
  break;

case 1:                                    /* Parse /proc/cpuinfo */
  arg2p=va_arg(ap,int*);
  arg2=*arg2p;
  if (arg2==0) weight=parse_cpuinfo();     /* Return bogomips, straight-up  */
  else {
    arg3p=va_arg(ap,int*);
    arg3=*arg3p;
    switch(arg3) {
    case 0:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"SHORT");
      break;                               /* Convert to HINT SHORT  */
    case 1:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"INT");
      break;                               /* Convert to HINT INT  */
    case 2:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"LONGLONG");
      break;                               /* Convert to HINT LONGLONG  */
    case 3:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"FLOAT");
      break;                               /* Convert to HINT FLOAT  */
    case 4:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"DOUBLE");
      break;                               /* Convert to HINT DOUBLE  */
    case 5:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"NAS");
      break;                               /* Convert to NAS NPB-W-serial  */
    default:
      weight=0.0;
    } /* switch (arg3) */
  } /* else (arg2!=0) */
  break;

case 2:                                    /* Calculate Pi & count flops */
  arg2p=va_arg(ap,int*);                   /* Level of precision */
  arg2=*arg2p;
  arg3p=va_arg(ap,int*);
  arg3=*arg3p;
  if (arg3==0) weight=calc_pi(pow(2,arg2),&pi);
                                           /* Return Mflops, straight-up  */
  else {
    arg4p=va_arg(ap,int*);
    arg4=*arg4p;
    switch(arg4) {
    case 0:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"SHORT");
      break;                               /* Convert to HINT SHORT  */
    case 1:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"INT");
      break;                               /* Convert to HINT INT  */
    case 2:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"LONGLONG");
      break;                               /* Convert to HINT LONGLONG  */
    case 3:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"FLOAT");
      break;                               /* Convert to HINT FLOAT  */
    case 4:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"DOUBLE");
      break;                               /* Convert to HINT DOUBLE  */
    case 5:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"NAS");
      break;                               /* Convert to NAS NPB-W-serial  */
      default:
        weight=0.0;
      } /* switch (arg4) */
    } /* else (arg3!=0) */
    break;

  case 3:                                    /* Use HINT b/m to weigh nodes */
    arg2p=va_arg(ap,int*);
    arg2=*arg2p;
    switch(arg2) {
    case 0:
      weight=run_hint("SHORT");
      break;                                 /* Convert to HINT SHORT  */
    case 1:
      weight=run_hint("INT");
      break;                                 /* Convert to HINT INT  */
    case 2:
      weight=run_hint("LONGLONG");
      break;                                 /* Convert to HINT LONGLONG  */
    case 3:
      weight=run_hint("FLOAT");
      break;                                 /* Convert to HINT FLOAT  */
    case 4:
      weight=run_hint("DOUBLE");
      break;                                 /* Convert to HINT DOUBLE  */
    default:
      weight=0.0;
    } /* switch (arg2) */
    break;

  case 4:                                    /* Use NAS NPB-W-serial */
    weight=0.0;                              /* Not yet available */
    break;

  default:
    weight=0.0;
  } /* switch (arg1) */

  va_end(ap);                                /* Wrap things up */
  return weight;
} /* weighnode_ */


double weighnode__(int *yardstick, ...) {
  /*********************************************************************
    See header for weighnode() for description & history
  *********************************************************************/
  double weight,*weightp,pi;
  int arg1,arg2,arg3,arg4;
  int *arg2p,*arg3p,*arg4p;

  va_list ap;
  va_start(ap,yardstick);                    /* Initialize variable arguments */

  arg1=*yardstick;
  switch(arg1) {

  case 0:                                    /* Let programmer specify weight */
    weightp=va_arg(ap,double*);              /* Return the second argument */
    weight=*weightp;
    break;

  case 1:                                    /* Parse /proc/cpuinfo */
    arg2p=va_arg(ap,int*);
    arg2=*arg2p;
    if (arg2==0) weight=parse_cpuinfo();     /* Return bogomips, straight-up  */
    else {
      arg3p=va_arg(ap,int*);
      arg3=*arg3p;
      switch(arg3) {
      case 0:
```

```
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"SHORT");
      break;                              /* Convert to HINT SHORT  */
    case 1:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"INT");
      break;                              /* Convert to HINT INT  */
    case 2:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"LONGLONG");
      break;                              /* Convert to HINT LONGLONG  */
    case 3:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"FLOAT");
      break;                              /* Convert to HINT FLOAT  */
    case 4:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"DOUBLE");
      break;                              /* Convert to HINT DOUBLE  */
    case 5:
      weight=convert_parse_cpuinfo(parse_cpuinfo(),"NAS");
      break;                              /* Convert to NAS NPB-W-serial  */
    default:
      weight=0.0;
    } /* switch (arg3) */
  } /* else (arg2!=0) */
  break;

case 2:                                   /* Calculate Pi & count flops */
  arg2p=va_arg(ap,int*);                  /* Level of precision */
  arg2=*arg2p;
  arg3p=va_arg(ap,int*);
  arg3=*arg3p;
  if (arg3==0) weight=calc_pi(pow(2,arg2),&pi);
                                          /* Return Mflops, straight-up  */
  else {
    arg4p=va_arg(ap,int*);
    arg4=*arg4p;
    switch(arg4) {
    case 0:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"SHORT");
      break;                              /* Convert to HINT SHORT  */
    case 1:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"INT");
      break;                              /* Convert to HINT INT  */
    case 2:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"LONGLONG");
      break;                              /* Convert to HINT LONGLONG  */
    case 3:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"FLOAT");
      break;                              /* Convert to HINT FLOAT  */
    case 4:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"DOUBLE");
      break;                              /* Convert to HINT DOUBLE  */
    case 5:
      weight=convert_calc_pi(calc_pi(pow(2,arg2),&pi),"NAS");
      break;                              /* Convert to NAS NPB-W-serial  */
    default:
      weight=0.0;
    } /* switch (arg4) */
  } /* else (arg3!=0) */
  break;

case 3:                                   /* Use HINT b/m to weigh nodes */
  arg2p=va_arg(ap,int*);
  arg2=*arg2p;
  switch(arg2) {
  case 0:
    weight=run_hint("SHORT");
    break;                                /* Convert to HINT SHORT  */
  case 1:
    weight=run_hint("INT");
    break;                                /* Convert to HINT INT  */
  case 2:
    weight=run_hint("LONGLONG");
    break;                                /* Convert to HINT LONGLONG  */
  case 3:
        weight=run_hint("FLOAT");
        break;                              /* Convert to HINT FLOAT  */
      case 4:
        weight=run_hint("DOUBLE");
        break;                              /* Convert to HINT DOUBLE  */
      default:
        weight=0.0;
      } /* switch (arg2) */
      break;

    case 4:                                 /* Use NAS NPB-W-serial */
      weight=0.0;                           /* Not yet available */
      break;

    default:
      weight=0.0;
    } /* switch (arg1) */

    va_end(ap);                             /* Wrap things up */
    return weight;
} /* weighnode__ */
```

C.2. metric

C.2.1. ***metric.h***

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/*********************************************************************
 *                                                                   *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous             *
 *          Cluster of PCs                                           *
 *          AFIT/GE/ENG/99M-02                                       *
 *                                                                   *
 * PACKAGE: NodeMetric                                               *
 *          0.2                                                      *
 *                                                                   *
 * FILE:    metric.h                                                 *
 *          0.1                                                      *
 *          Measures the performance of a node and returns an        *
 *          appropriate weight                                       *
 *                                                                   *
 * AUTHOR:  Capt Christopher A. Bohn                                 *
 *                                                                   *
 * HISTORY: 16 Nov 98 -- Version a.1 begun                           *
 *          18 Nov 98 -- a.1 complete                                *
 *          19 Nov 98 -- Version a.2 begun                           *
 *                    -- a.2 complete                                *
 *          20 Nov 98 -- Version a.3 begun                           *
 *                    -- a.3 abandoned; version a.3.1 begun          *
 *                    -- a.3.1 complete                              *
 *          23 Nov 98 -- Version a.3.2 begun -- Objective: see above *
 *          24 Nov 98 -- a.3.2 complete                              *
 *                    -- Version a.3.3 begun                         *
 *                    -- a.3.3 complete                              *
 *          25 Nov 98 -- metric.h Version 0.1                        *
 *          29 Nov 98 -- NodeMetric Version 0.1                      *
 *          11 Dec 98 -- NodeMetric 0.1.1                            *
 *          31 Dec 98 -- NodeMetric 0.1.2                            *
 *           1 Jan 99 -- NodeMetric 0.2                              *
 *                                                                   *
 *********************************************************************/


double parse_cpuinfo ();
double calc_pi (long,double*);
double run_hint (char[]);



/* parse_cpuinfo */
  /******************************************************************
    Parses /proc/cpuinfo.  Returns bogomips if /proc/cpuinfo exists,
    0.0 otherwise.  For now, we're only looking at bogomips; neglect cpu,
    model, vendor_id ... also assume uniprocessor.
      MIPS, of course, is "Million Instructions Per Second" (or, if you
    prefer, "Meaningless Indicator of Performance Standard"), and BOGO
    is a prefix to indicate bogusness -- it's only a calculated guess.
      One big advantage to this benchmark is it's cheap.  No calculations
    to be performed!  Just parse in a file that doesn't even exist on disk.
    If it does exist, then it resides in core memory!
  ******************************************************************/



/* calc_pi */
  /******************************************************************
    Calculates Pi by estimating the area under a curve.  Returns the
    number of Millions of FLoating Point Operations per Second for the
    kernel.  Accepts as a parameter the number of intervals in which to
    divide the curve for the integration.  Passes back a reference to the
    calculated value of Pi; this is necessary, or a good optimizing compiler
    will avoid calculating Pi at all, since it would never be used.  This
    version only works for n < (2**31-1)/6.  That's just as well, since this
    is intended to be a "quick'n'dirty" benchmark.
  *********************************************************************/


/* run_hint */
  /*********************************************************************
    Runs the HINT benchmark to evaluate the system's performance.
    Specify the datatype to be evaluated as the parameter.
      Returns the QUIPS value provided by HINT.
      Assumes the compiled HINT executables are stored in a directory
    called "hint" immediately below the current directory.
  *********************************************************************/
```

C.2.2. ***metric.c***

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/*********************************************************************
 *                                                                   *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous             *
 *          Cluster of PCs                                           *
 *          AFIT/GE/ENG/99M-02                                       *
 *                                                                   *
 * PACKAGE: NodeMetric                                               *
 *          0.2                                                      *
 *                                                                   *
 * FILE:    metric.c                                                 *
 *          0.1                                                      *
 *          Measures the performance of a node and returns an        *
 *          appropriate weight                                       *
 *                                                                   *
 * AUTHOR:  Capt Christopher A. Bohn                                 *
 *                                                                   *
 * HISTORY: 16 Nov 98 -- Version a.1 begun                           *
 *                       -- Objective: parse /proc/cpuinfo           *
 *          18 Nov 98 -- Successfully parsed /proc/cpuinfo           *
 *                       -- Does not check for EOF--implement later   *
 *                    -- a.1 complete                                *
 *          19 Nov 98 -- Version a.2 begun                           *
 *                       -- Objective: time the calculation of pi     *
 *                    -- Successfully calculted Pi, and experimented  *
 *                       with a couple different datatype sizes.      *
 *                       -- Conclusion: The intrinsic datatypes large *
 *                          enough to deal with numbers > 2**31 don't *
 *                          matter here, because the execution takes  *
 *                          so long as to defeat the point of the pi  *
 *                          metric -- something that takes a half-hour*
 *                          to finish might as well lead us to use a  *
 *                          "real" benchmark like HINT -- the pi b/m  *
 *                          is intended to be something fast that is  *
 *                          not as naive as the cpuinfo b/m           *
 *                    -- a.2 complete                                *
 *          20 Nov 98 -- Version a.3 begun                           *
 *                       -- Objective: fork another process that will *
 *                          transmorgriphy into the serial HINT b/m   *
 *                    -- a.3 abandoned; version a.3.1 begun           *
 *                       -- Objective: take core of a.3's run_hint,   *
 *                          and instead of forking and trying to      *
 *                          assess the status of the child process,   *
 *                          blahblahblah, we'll have the parent       *
 *                          process use a system call to run the b/m, *
 *                          and the parent will stay blocked until    *
 *                          the b/m is complete.                      *
 *                    -- Successfully launched HINT -- now just need  *
 *                       to add some flexibility that will allow      *
 *                       multiple processors to run it without        *
 *                       overwriting each other's "stuff"             *
 *                    -- a.3.1 complete                              *
 *          23 Nov 98 -- Version a.3.2 begun -- Objective: see above  *
 *          24 Nov 98 -- a.3.2 complete                              *
 *                    -- Version a.3.3 begun                         *
 *                       -- Objective: change parse_cpuinfo to return *
 *                          bogomips instead of passing it as a       *
 *                          parameter                                 *
 *                    -- a.3.3 complete                              *
 *          25 Nov 98 -- Testing constructs removed                  *
 *                    -- metric.c Version 0.1                        *
 *          29 Nov 98 -- NodeMetric Version 0.1                      *
 *          11 Dec 98 -- NodeMetric 0.1.1                            *
 *          31 Dec 98 -- NodeMetric 0.1.2                            *
 *          1 Jan 99 -- NodeMetric 0.2                                   *
 *                                                                       *
 ***********************************************************************/


double parse_cpuinfo () {
  /*********************************************************************
    Parses /proc/cpuinfo.  Returns bogomips if /proc/cpuinfo exists,
    0.0 otherwise.  For now, we're only looking at bogomips; neglect cpu,
    model, vendor_id ... also assume uniprocessor.
      MIPS, of course, is "Million Instructions Per Second" (or, if you
    prefer, "Meaningless Indicator of Performance Standard"), and BOGO
    is a prefix to indicate bogusness -- it's only a calculated guess.
      This benchmark is remarkably naive.  There is no way we could hope
    to realize the level of performance bogomips indicates.  And the memory
    hierarchy tends to level the field -- a node with a 600MHz processor
    isn't twice as fast as a node with a 300MHz processor.  But a table
    lookup might be able to account for this.  Caution, though:  interpolating
    between & extrapolating from known values in the table could be dangerous,
    as that might not account for changes in memory speed &/or bandwidth.
    Another issue to ponder is that there are processors for which the
    bogomips does not correspond to the clock speed.  Natch, in an unpipelined
    nonsupercalar processor, this is to be expected ... but what about the
    i80486?  A problem for another day.
      One big advantage to this benchmark is it's cheap.  No calculations
    to be performed!  Just parse in a file that doesn't even exist on disk.
    If it does exist, then it resides in core memory!
    16 Nov 98 -- dabbled
    18 Nov 98 -- successful parse of /proc/cpuinfo, passing back bogomips
              -- successful return of error code if file does not exist,
                 i.e., this is not a Linux system
    24 Nov 98 -- Instead of returning -1 for error and 0 for success, and
                 passing bogomips as a parameter, it now takes no parameters
                and returns bogomips if successful and 0.0 if not
  *********************************************************************/
  FILE  *cpuinfo;
  char line[31] = "";                     /* More than sufficient */
  double bogomips;

  cpuinfo = fopen("/proc/cpuinfo","r");

  if (cpuinfo == NULL)                    /* File does not exist */
    return(0.0);

  else {
    while ( strcmp(line,"bogomips") ) {   /* Until we find the target */
      fscanf(cpuinfo,"%s",line);            /* I suppose I oughtta check */
    } /* while ( strcmp ) */                /* for EOF ... later */

    fscanf(cpuinfo,"%s",line);            /* Should be a colon */
    fscanf(cpuinfo,"%s",line);            /* The magic number we want */
    bogomips = atof(line);
    fclose(cpuinfo);
    return(bogomips);
  } /* else (cpuinfo != NULL) */
} /* parse_cpuinfo */


double calc_pi (long n, double *globalpi) {
  /*********************************************************************
    Calculates Pi by estimating the area under a curve.  Returns the
    number of Millions of FLoating Point Operations per Second for the
    kernel.  Accepts as a parameter the number of intervals in which to
    divide the curve for the integration.  Passes back a reference to the
    calculated value of Pi; this is necessary, or a good optimizing compiler
    will avoid calculating Pi at all, since it would never be used.  This
    version only works for n < (2**31-1)/6.  That's just as well, since this
    is intended to be a "quick'n'dirty" benchmark.
      Like the cpuinfo benchmark, the pi benchmark is naive, only not so
    much.  It actually does /some/ work, but not enough to break out of
    the L1 cache, or even the register file.  This, too, should require
    a table look-up to map the produced results into a useful weight.
      It is interesting to note that peak Mflops is at or about n=2**17 or 18
    on the Pentium II boxen I'm developing & testing this on (not including
    the "infinity" values for really small values of n), but it holds a
    more-or-less stable value for larger values of n.  As a point of
    reference, once out of the "infinity" stage (at n=2**14), the Mflops
    rating grows up to the peak.  Presumably, the beast cannot be sufficiently
    fed for n less than 2**17.  I might want to ponder why it peaks and then
    stablizes.
    19 Nov 98 -- borrowed from cpi.c found in the MPICH 1.1 distro,
                 removed parallelism from it, minimized it, added
                Mflops calculation
  *******************************************************************/
  double pi = *globalpi;
  clock_t starttime, endtime;
  long i, flopcount;
  double h, x, sum, mflops, totaltime;

  starttime = clock();

  h = 1.0 / (double)n;                  /* 1 flop */
  sum = 0.0;                            /* 0 flop */
  for (i=0; i<n; i++) {                 /* these are all integer ops */
    x = h * ((double)i - 0.5);          /* 2 flop */
    sum += (4.0 / (1.0 + x*x));         /* 4 flop */
  }
  pi = h * sum;                         /* 1 flop */

  endtime = clock();

  totaltime = (double)(endtime-starttime) / (double)CLOCKS_PER_SEC;
  flopcount = 6*n+2;
  mflops = ( (double)flopcount / 1000000.0 ) / totaltime;

  *globalpi = pi;
  return(mflops);
} /* calc_pi */



double run_hint (char datatype[]) {
  /*******************************************************************
    Runs the HINT benchmark to evaluate the system's performance.
    Specify the datatype to be evaluated as the parameter.
      Returns the QUIPS value provided by HINT.
      Assumes the compiled HINT executables are stored in a directory
    called "hint" immediately below the current directory.
    20 Nov 98 -- Much experimenting with fork() & execv(), abandoned
              -- Used system() to launch HINT, with run_hint() waiting
                 for HINT to complete before progressing
    23 Nov 98 -- Accomplished much (but not all) generalization (all
                 except the parsing section and testing)
    24 Nov 98 -- Discovered (duh!) that system() does not return the
                 output of the call (e.g., "pwd"), and that
                system("cd ...") does not effect a permanent change
                of working directory (nuts!)
                -- Going to have to let the multiple copies of HINT
                   overwrite each other's output into the ./data
                   directory ... 'sokay, since I'm interested in what
                   HINT places on stdout ... just hope they don't
                   crash -- eliminates need for localpath parameter
  *******************************************************************/
  char command[61] = "mkdir ";
  FILE *hintout;
  char line[401] = "";                /* Enough for five lines of dots */
  double quips;

  switch (datatype[0]) {              /* Fix the datatype variable to */
                                        /* match the executable name */
                                        /* (don't trust the end user) */
  case 's':                           /* Assume "SHORT" */
  case 'S':
    datatype="SHORT";
    break;
  case 'i':                               /* Assume "INT" */
  case 'I':
    datatype="INT";
    break;
  case 'f':                               /* Assume "FLOAT" */
  case 'F':
    datatype="FLOAT";
    break;
  case 'l':                               /* Assume "LONG LONG" or */
  case 'L':                                 /* "LONGLONG" but not "LONG" */
    datatype="LONGLONG";
    break;
  default:
  case 'd':                               /* Assume "DOUBLE" */
  case 'D':
    datatype="DOUBLE";
  } /* switch (datatype[0]) */

  strcpy(command,"mkdir data");           /* Create appropriate directories */
  system(command);                          /* if they don't exist */
  strcat(command,"/hint");                /* This wouldn't have been needed */
  system(command);                          /* if I was able to change */
                                            /* the working directory! */
                                          /* We'll accept that this will */
                                            /* return errors for most (or */
                                            /* all) of the processes */

  strcpy(command,"hint/");                /* Launch HINT ... */
  strcat(command,datatype);
  hintout = popen(command,"r");           /* ... and take its stdout */

  while ( strcmp(line,"Finished") )       /* Until we find the target */
    fscanf(hintout,"%s",line);

  fscanf(hintout,"%s",line);              /* Should be "with" */
  fscanf(hintout,"%s",line);              /* The magic number we want */
  quips = atof(line);

  pclose(hintout);
  return quips;
} /* run_hint */
```

C.3. metricmap

C.3.1. metricmap.h

```
/************************************************************************
 *                                                                      *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous                *
 *          Cluster of PCs                                              *
 *          AFIT/GE/ENG/99M-02                                          *
 *                                                                      *
 * PACKAGE: NodeMetric                                                  *
 *          0.2                                                         *
 *                                                                      *
 * FILE:    metricmap.h                                                 *
 *          0.1                                                         *
 *          Converts skewed performance weights into better weights     *
 *                                                                      *
 * AUTHOR:  Capt Christopher A. Bohn                                    *
 *                                                                      *
 * HISTORY: 25 Nov 98 -- Version a.1 begun                              *
 *                    -- a.1 complete                                   *
 *                    -- Version a.2 begun                              *
 *                    -- a.2 complete                                   *
 *                    -- Version a.3 begun                              *
 *                    -- a.3 complete                                   *
 *          27 Nov 98 -- Version a.4 begun                              *
 *                    -- a.4 complete                                   *
 *          29 Nov 98 -- Version a.5 begun                              *
 *                    -- a.5 complete                                   *
 *                    -- metricmap.h Version 0.1                        *
 *                    -- NodeMetric Version 0.1                         *
 *          11 Dec 98 -- Nodemetric 0.1.1                               *
 *          31 Dec 98 -- NodeMetric 0.1.2                               *
 *           1 Jan 99 -- NodeMetric 0.2                                 *
 *                                                                      *
 ************************************************************************/


double convert_parse_cpuinfo (double,char[]);
double convert_calc_pi (double,char[]);


/* convert_parse_cpuinfo */
  /*********************************************************************
    Converts the bogomips returned from parse_cpuinfo into a different
    weight.  First argument is the value returned by parse_cpuinfo.
    Second argument establishes how the value should be changed.
         S -- based on HINT SHORT metric
         I -- based on HINT INT metric
         L -- based on HINT LONGLONG metric
         F -- based on HINT FLOAT metric
         D -- based on HINT DOUBLE metric
           -- "none of the above" returns bogomips unchanged
  *********************************************************************/


/* convert_calc_pi */
  /*********************************************************************
    Converts the mflops returned from calc_pi into a different
    weight.  First argument is the value returned by calc_pi.
    Second argument establishes how the value should be changed.
         S -- based on HINT SHORT metric
         I -- based on HINT INT metric
         L -- based on HINT LONGLONG metric
         F -- based on HINT FLOAT metric
         D -- based on HINT DOUBLE metric
           -- "none of the above" returns mflops unchanged
  *********************************************************************/
```

C.3.2. metricmap.c

```
#include <math.h>
#include <stdio.h>
#include "nodeinfo.h"
/*********************************************************************
 *                                                                   *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous             *
 *          Cluster of PCs                                           *
 *          AFIT/GE/ENG/99M-02                                       *
 *                                                                   *
 * PACKAGE: NodeMetric                                               *
 *          0.2                                                      *
 *                                                                   *
 * FILE:    metricmap.c                                              *
 *          0.2                                                      *
 *          Converts skewed performance weights into better weights  *
 *                                                                   *
 * AUTHOR:  Capt Christopher A. Bohn                                 *
 *                                                                   *
 * HISTORY: 25 Nov 98 -- Version a.1 begun                           *
 *                       -- Objective: Develop interface to convert  *
 *                          bogomips & pi-determined Mflops into     *
 *                          useful weights -- interface will be      *
 *                          independent of implementation --         *
 *                          if implementation is changed, no need to *
 *                          do full recompilation, just recompile    *
 *                          implementation & relink (just be sure to *
 *                          type "make" /before/ you update the      *
 *                          version history in metricmap.h, or the   *
 *                          dependency on metricmap.h will force     *
 *                          the calling program to recompile         *
 *                    -- a.1 complete                                *
 *                    -- Version a.2 begun                           *
 *                       -- Objective: q&d implementation of         *
 *                          convert_cpuinfo -- IF-THEN-ELSE          *
 *                          constructs should be sufficient at this  *
 *                          point -- as the map space grows, an      *
 *                          actual data structure would probably be a *
 *                          good idea, from a maintenance and        *
 *                          cleanliness POV -- structure only at this *
 *                          point, since I have no values to plug in *
 *                          yet                                      *
 *                    -- a.2 complete                                *
 *                    -- Version a.3 begun                           *
 *                       -- Objective: same as a.2, except for       *
 *                          convert_calc_pi                          *
 *                    -- a.3 complete                                *
 *          27 Nov 98 -- Version a.4 begun                           *
 *                       -- Objective: provide specific values for   *
 *                          333MHz & 400MHz Pentium II's             *
 *                    -- a.4 complete                                *
 *          29 Nov 98 -- Version a.5 begun                           *
 *                       -- Objective: if passed 0.0 (the designated *
 *                          error weight), return 0.0                *
 *                    -- a.5 complete                                *
 *                    -- metricmap.c Version 0.1                     *
 *                    -- NodeMetric Version 0.1                      *
 *          11 Dec 98 -- Nodemetric 0.1.1                            *
 *          31 Dec 98 -- NodeMetric 0.1.2                            *
 *                    -- Version 0.2 begun                           *
 *                       -- Objective: remove hard-coded mapping and *
 *                          use the maps generated by buildmap       *
 *                    -- metricmap.c 0.2                             *
 *                       -- After buildmap is finished executing,    *
 *                          I'll test metricmap 0.2, and if all goes *
 *                          well, /then/ I'll declare NodeMetric 0.2 *
 *           1 Jan 99 -- NodeMetric 0.2                              *
 *                                                                   *
 *********************************************************************/
```

```
double convert_parse_cpuinfo (double bogomips,char factor[]) {
  /******************************************************************
    Converts the bogomips returned from parse_cpuinfo into a different
    weight.  First argument is the value returned by parse_cpuinfo.
    Second argument establishes how the value should be changed.
         S -- based on HINT SHORT metric
         I -- based on HINT INT metric
         L -- based on HINT LONGLONG metric
         F -- based on HINT FLOAT metric
         D -- based on HINT DOUBLE metric
           -- "none of the above" returns bogomips unchanged
    25 Nov 98 -- This implementation uses an IF-THEN-ELSE construct --
                 future implementations should use a data structure for
                 maintainability -- also need to think about how to deal
                with unexpected values (return closest value, inter/extra-
                polate?)
                 -- Basic structure; still need to get actual values to
                   return
    27 Nov 98 -- Incorporate numbers for 333MHz & 400MHz Pentium II's
    29 Nov 98 -- Added a check for 0.0
    31 Dec 98 -- Removed hard-coded map -- make use of dynamic map
              -- Linearly interpolate between known values if need be
                 -- Extrapolate as follows:
                 -- Less than lowest known value, interpolate with zero
                    (provided in map)
                 -- Greater than largest known value, use largest known
                    value (do not extrapolate, especially to with
                   infinity :> )
  ******************************************************************/

  FILE *mipsfile;
  nodeinfo *mipslist;
  int i;
  double lower,upper,lo,hi,rise,run,diff;

  mipsfile  = fopen(filename1,"r");
  if ( mipsfile == NULL ) {                /* File does not exist */
    return bogomips;                       /* Default action is no action */
  } /* if ( mipsfile == NULL ) */
  else {                                   /* File does exist */
    mipslist = load(mipsfile);
    fclose(mipsfile);

    lower = upper = 0.0; i = 0;
    while ( ( upper < bogomips) && ( i < mipslist[0].listsize  ) ) {
      lower = upper;
      upper = mipslist[++i].key;
    } /* while ( ( upper < bogomips ) && ( i < mipslist[0].listsize  ) ) */

    if ( upper == bogomips ) {             /* Straight-forward map */
      if      ( factor[0]=='S' || factor[0]=='s' ) return mipslist[i].H_short;
      else if ( factor[0]=='I' || factor[0]=='i' ) return mipslist[i].H_int;
      else if ( factor[0]=='L' || factor[0]=='l' ) return mipslist[i].H_long;
      else if ( factor[0]=='F' || factor[0]=='f' ) return mipslist[i].H_float;
      else if ( factor[0]=='D' || factor[0]=='d' ) return mipslist[i].H_double;
      else                                         return bogomips;
    } /* if ( upper == bogomips ) */
    else {                                 /* Need to inference */
      if ( i == mipslist[0].listsize ) {   /* We reached the largest known */
                                           /* value and it's too small */
        if      ( factor[0]=='S' || factor[0]=='s' ) return mipslist[i].H_short;
        else if ( factor[0]=='I' || factor[0]=='i' ) return mipslist[i].H_int;
        else if ( factor[0]=='L' || factor[0]=='l' ) return mipslist[i].H_long;
        else if ( factor[0]=='F' || factor[0]=='f' ) return mipslist[i].H_float;
        else if ( factor[0]=='D' || factor[0]=='d' ) return mipslist[i].H_double;
        else                                         return bogomips;
      } /* ( i == mipslist[0].listsize ) */
      else {                               /* Interpolate */
        run = upper-lower;
        diff = bogomips-lower;
        if      ( factor[0]=='S' || factor[0]=='s' ) {
```

```
        hi = mipslist[i].H_short;
        lo = mipslist[i-1].H_short;
        rise = hi - lo;
        return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
      } /* HINT SHORT */
      else if ( factor[0]=='I' || factor[0]=='i' ) {
        hi = mipslist[i].H_int;
        lo = mipslist[i-1].H_int;
        rise = hi - lo;
        return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
      } /* HINT INT */
      else if ( factor[0]=='L' || factor[0]=='l' ) {
        hi = mipslist[i].H_long;
        lo = mipslist[i-1].H_long;
        rise = hi - lo;
        return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
      } /* HINT LONG */
      else if ( factor[0]=='F' || factor[0]=='f' ) {
        hi = mipslist[i].H_float;
        lo = mipslist[i-1].H_float;
        rise = hi - lo;
        return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
      } /* HINT FLOAT */
      else if ( factor[0]=='D' || factor[0]=='d' ) {
        hi = mipslist[i].H_double;
        lo = mipslist[i-1].H_double;
        rise = hi - lo;
        return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
      } /* HINT DOUBLE */
      else /* none of the above */
        return bogomips;
     } /* else ( i < mipslist[0].listsize ) */
    } /* else ( upper > bogomips ) */
  } /* else ( mipsfile != NULL ) */
} /* convert_parse_cpuinfo */


double convert_calc_pi (double mflops,char factor[]) {
  /*********************************************************************
    Converts the mflops returned from calc_pi into a different
    weight.  First argument is the value returned by calc_pi.
    Second argument establishes how the value should be changed.
         S -- based on HINT SHORT metric
        I -- based on HINT INT metric
        L -- based on HINT LONGLONG metric
        F -- based on HINT FLOAT metric
        D -- based on HINT DOUBLE metric
          -- "none of the above" returns mflops unchanged
    25 Nov 98 -- This implementation uses an IF-THEN-ELSE construct --
                 future implementations should use a data structure for
                 maintainability -- also need to think about how to deal
                with unexpected values (return closest value, inter/extra-
                polate?)
                 -- Basic structure; still need to get actual values to
                   return
    27 Nov 98 -- Incorporate numbers for 333MHz & 400MHz Pentium II's
    29 Nov 98 -- Added a check for 0.0
    31 Dec 98 -- Removed hard-coded map -- make use of dynamic map
              -- Linearly interpolate between known values if need be
                 -- Extrapolate as follows:
                 -- Less than lowest known value, interpolate with zero
                    (provided in map)
                 -- Greater than largest known value, use largest known
                    value (do not extrapolate, especially to with
                   infinity :> )
  *********************************************************************/

  FILE *flopsfile;
  nodeinfo *flopslist;
  int i;
  double lower,upper,lo,hi,rise,run,diff;
  flopsfile  = fopen(filename2,"r");
  if ( flopsfile == NULL ) {                /* File does not exist */
    return mflops;                          /* Default action is no action */
  } /* if ( flopsfile == NULL ) */
  else {                                    /* File does exist */
    flopslist = load(flopsfile);
    fclose(flopsfile);

    lower = upper = 0.0; i = 0;
    while ( ( upper < mflops) && ( i < flopslist[0].listsize  ) ) {
      lower = upper;
      upper = flopslist[++i].key;
    } /* while ( ( upper < mflops ) && ( i < flopslist[0].listsize  ) ) */

    if ( upper == mflops ) {                  /* Straight-forward map */
      if      ( factor[0]=='S' || factor[0]=='s' ) return flopslist[i].H_short;
      else if ( factor[0]=='I' || factor[0]=='i' ) return flopslist[i].H_int;
      else if ( factor[0]=='L' || factor[0]=='l' ) return flopslist[i].H_long;
      else if ( factor[0]=='F' || factor[0]=='f' ) return flopslist[i].H_float;
      else if ( factor[0]=='D' || factor[0]=='d' ) return flopslist[i].H_double;
      else                                         return mflops;
    } /* if ( upper == mflops ) */
    else {                                  /* Need to inference */
      if ( i == flopslist[0].listsize ) {   /* We reached the largest known */
                                            /* value and it's too small */
        if      ( factor[0]=='S' || factor[0]=='s' ) return flopslist[i].H_short;
        else if ( factor[0]=='I' || factor[0]=='i' ) return flopslist[i].H_int;
        else if ( factor[0]=='L' || factor[0]=='l' ) return flopslist[i].H_long;
        else if ( factor[0]=='F' || factor[0]=='f' ) return flopslist[i].H_float;
        else if ( factor[0]=='D' || factor[0]=='d' ) return flopslist[i].H_double;
        else                                         return mflops;
      } /* ( i == flopslist[0].listsize ) */
      else {                                /* Interpolate */
        run = upper-lower;
        diff = mflops-lower;
        if      ( factor[0]=='S' || factor[0]=='s' ) {
          hi = flopslist[i].H_short;
          lo = flopslist[i-1].H_short;
          rise = hi - lo;
          return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
        } /* HINT SHORT */
        else if ( factor[0]=='I' || factor[0]=='i' ) {
          hi = flopslist[i].H_int;
          lo = flopslist[i-1].H_int;
          rise = hi - lo;
          return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
        } /* HINT INT */
        else if ( factor[0]=='L' || factor[0]=='l' ) {
          hi = flopslist[i].H_long;
          lo = flopslist[i-1].H_long;
          rise = hi - lo;
          return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
        } /* HINT LONG */
        else if ( factor[0]=='F' || factor[0]=='f' ) {
          hi = flopslist[i].H_float;
          lo = flopslist[i-1].H_float;
          rise = hi - lo;
          return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
        } /* HINT FLOAT */
        else if ( factor[0]=='D' || factor[0]=='d' ) {
          hi = flopslist[i].H_double;
          lo = flopslist[i-1].H_double;
          rise = hi - lo;
          return fabs(lo+diff*rise/run);    /* fabs() just to be safe */
        } /* HINT DOUBLE */
        else /* none of the above */
          return mflops;
      } /* else ( i < flopslist[0].listsize ) */
    } /* else ( upper > mflops ) */
  } /* else ( flopsfile != NULL ) */
} /* convert_calc_pi */
```

C.4. buildmap.c

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "metric.h"
#include "nodeinfo.h"

/**********************************************************************
 *                                                                    *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous              *
 *          Cluster of PCs                                            *
 *          AFIT/ENG/GE99M-02                                         *
 *                                                                    *
 * PACKAGE: NodeMetric                                                *
 *          0.2                                                       *
 *                                                                    *
 * FILE:    buildmap.c                                                *
 *          0.1                                                       *
 *          Builds the maps used by metricmap.c                       *
 *                                                                    *
 * AUTHOR:  Capt Christopher A. Bohn                                  *
 *                                                                    *
 * HISTORY: 28 Dec 98 -- Version a.1 begun                            *
 *                       -- Objective: basic  implementation --       *
 *                       -- Basic  structure in place -- still        *
 *                          need to get the struct nodeinfo material  *
 *                          working                                   *
 *                          -- Passes the "egcs -c buildmap.c" test   *
 *                    -- a.1 complete                                 *
 *          29 Dec 98 -- Version a.2 begun                            *
 *                       -- Objective: finish struct nodeinfo-related *
 *                          material                                  *
 *                          -- Moving the struct nodeinfo definition  *
 *                             from buildmap.c to nodeinfo.h          *
 *                             -- metricmap.c will also need it       *
 *                    -- a.2 complete                                 *
 *                    -- Version a.3 begun                            *
 *                       -- Objective: write front-end                *
 *                    -- a.3 complete; won't declare Version 0.1 until*
 *                       I write load() & save() (in another file)    *
 *                       -- for that matter, I still gotta test       *
 *          30 Dec 98 -- Version a.4 begun                            *
 *                       -- Objective: cleaning-up / deobfuscation    *
 *                    -- a.4 complete                                 *
 *                    -- The bloody thing compiles & links (woo-hoo!),*
 *                       but for initial testing, I'm going to disable*
 *                       the HINT portion, since they, um, take a     *
 *                       really, really long time                     *
 *                       -- Yee-haw!  Core dump.  Debugging time.     *
 *          31 Dec 98 -- Version a.5 begun                            *
 *                       -- Objective: fix bar() right after I fix    *
 *                          foo()                                     *
 *                          -- I must really need a good night's      *
 *                             sleep ... there's a big problem with   *
 *                             the data structure                     *
 *                          -- I'm going to recode this as a linear   *
 *                             list (for now) ... it's a VERY simple  *
 *                             data structure, and for my initial     *
 *                             tests, at least, the time-complexity   *
 *                             (for small 'n') of O(n) should be      *
 *                              unappreciable                         *
 *                    -- a.5 complete                                 *
 *                    -- Still trying to track down that segmentation *
 *                       fault                                        *
 *                       are now written identically, and improperly  *
 *                       -- Doh!  I'm allocating memory for the lists *
 *                          in the wrong part of the program!         *
 *                          -- Move preload() into load() & allocate  *
 *                               there                                  *
 *                    -- buildmap.c 0.1                                 *
 *                    -- NodeMetric 0.1.2                               *
 *           1 Jan 99 -- NodeMetric 0.2                                 *
 *                                                                      *
 ************************************************************************/

void initialize (nodeinfo **);
void add_node (nodeinfo **, nodeinfo);
int compare (void *, void *);

#define min(x,y)  (((x)<(y))?(x):(y))
#define max(x,y)  (((x)>(y))?(x):(y))


int main () {
  FILE *mipsfile,*flopsfile;
  nodeinfo *mipslist,*flopslist;
  nodeinfo mipsnode,flopsnodelo,flopsnodehi;
  double H_short,H_int,H_long,H_float,H_double;
  time_t time1,time2;
  int i;
  double pi,mflops;

  mipsfile  = fopen(filename1,"r");
  flopsfile = fopen(filename2,"r");
  if ( ( mipsfile == NULL ) || ( flopsfile == NULL ) ) {
    initialize(&mipslist);                /* Files do not exist */
    initialize(&flopslist);
  } /* if ( ( mipsfile == NULL ) || ( flopsfile == NULL ) ) */
  else {                                  /* Files do exist */
    mipslist     = load(mipsfile);
    flopslist    = load(flopsfile);
    fclose(mipsfile);fclose(flopsfile);
  } /* else ( ( mipsfile != NULL ) && ( flopsfile != NULL ) ) */

  /* assess current node */
  time1 = time(NULL);
  mipsnode.key = parse_cpuinfo();
  time2 = time(NULL);
  printf("%d sec required to read bogomips.\n",
        (int)difftime(time2,time1));
  flopsnodelo.key = 1000000.0;            /* initialize for max/min */
  flopsnodehi.key =       0.0;            /* initialize for max/min */
  for ( i = 21 ; i < 28 ; i++) {
    time1 = time(NULL);
    mflops = calc_pi(pow(2,i),&pi);
    time2 = time(NULL);
    flopsnodelo.key = min(flopsnodelo.key,mflops);
    flopsnodehi.key = max(flopsnodelo.key,mflops);
    printf("%d sec required to calculate pi at precision level %d.\n",
          (int)difftime(time2,time1),i);
  } /* for i */
  printf("<< Starting HINT SHORT benchmark >>\n");
  time1 = time(NULL);
  H_short  = run_hint("SHORT");
  time2 = time(NULL);
  printf("%fmin required to complete HINT SHORT benchmark.\n",
        difftime(time2,time1)/60.0);
  printf("<< Starting HINT INT benchmark >>\n");
  time1 = time(NULL);
  H_int    = run_hint("INT");
  time2 = time(NULL);
  printf("%fmin required to complete HINT INT benchmark.\n",
        difftime(time2,time1)/60.0);
  printf("<< Starting HINT LONG benchmark >>\n");
  time1 = time(NULL);
  H_long   = run_hint("LONG");
  time2 = time(NULL);
  printf("%fmin required to complete HINT LONG benchmark.\n",
        difftime(time2,time1)/60.0);
  printf("<< Starting HINT FLOAT benchmark >>\n");
  time1 = time(NULL);
  H_float  = run_hint("FLOAT");
  time2 = time(NULL);
  printf("%fmin required to complete HINT FLOAT benchmark.\n",
         difftime(time2,time1)/60.0);
  printf("<< Starting HINT DOUBLE benchmark >>\n");
  time1 = time(NULL);
  H_double = run_hint("DOUBLE");
  time2 = time(NULL);
  printf("%fmin required to complete HINT DOUBLE benchmark.\n",
         difftime(time2,time1)/60.0);
  mipsnode.H_short  = flopsnodelo.H_short  = flopsnodehi.H_short  = H_short;
  mipsnode.H_int    = flopsnodelo.H_int    = flopsnodehi.H_int    = H_int;
  mipsnode.H_long   = flopsnodelo.H_long   = flopsnodehi.H_long   = H_long;
  mipsnode.H_float  = flopsnodelo.H_float  = flopsnodehi.H_float  = H_float;
  mipsnode.H_double = flopsnodelo.H_double = flopsnodehi.H_double = H_double;

  /* add info on current node to lists */
  add_node(&mipslist,mipsnode);
  add_node(&flopslist,flopsnodelo);
  add_node(&flopslist,flopsnodehi);

  mipsfile  = fopen(filename1,"w");
  flopsfile = fopen(filename2,"w");
  save(mipsfile,mipslist);
  save(flopsfile,flopslist);
  fclose(mipsfile);fclose(flopsfile);

  return 0;
} /* main() */


void initialize (nodeinfo *A[]) {
  (*A) = calloc(3,sizeof(nodeinfo));
  (*A)[0].listsize = 0;
  (*A)[0].key = 0.0;
  (*A)[0].H_short  = 0.0;
  (*A)[0].H_int    = 0.0;
  (*A)[0].H_long   = 0.0;
  (*A)[0].H_float  = 0.0;
  (*A)[0].H_double = 0.0;
} /* initialize() */


void add_node (nodeinfo *A[], nodeinfo node) {
  int listsize = (*A)[0].listsize + 1;
  (*A)[listsize] = node;
  qsort((*A), (listsize+1), sizeof(nodeinfo), compare);
  (*A)[0].listsize = listsize;
} /* add_node() */


int compare (void *va, void *vb) {
  /* was going to just return *a.key-*b.key, but that's a real, so then I
     was going to return (int)(*a.key-*b.key), but the truncation of 0.x or
     -0.x would provide invalid results, so... */
  nodeinfo *a=va, *b=vb;
  return (((*a).key < (*b).key) ? -1 : (((*a).key > (*b).key) ? 1 : 0));
} /* compare() */
```

C.5. nodeinfo.h

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

/**********************************************************************
 *                                                                    *
 * PROJECT: Asymmetric Load Balancing on a Heterogeneous              *
 *          Cluster of PCs                                            *
 *          AFIT/ENG/GE99M-02                                         *
 *                                                                    *
 * PACKAGE: NodeMetric                                                *
 *          0.2                                                       *
 *                                                                    *
 * FILE:    nodeinfo.h                                                *
 *          0.1                                                       *
 *          Key parts for obtaining & storing node metric information *
 *                                                                    *
 * AUTHOR:  Capt Christopher A. Bohn                                  *
 *                                                                    *
 * HISTORY: 28 Dec 98 -- buildmap.c a.1 has struct nodeinfo           *
 *          29 Dec 98 -- Version a.1 begun                            *
 *                       -- Objective: finish struct nodeinfo         *
 *                          definition                                *
 *                       -- Moved the struct nodeinfo definition from *
 *                          buildmap.c to nodeinfo.h                  *
 *                          -- metricmap.c will also need it          *
 *                       -- Prototyped load() & save()                *
 *                    -- a.1 complete                                 *
 *          30 Dec 98 -- Version a.2 begun                            *
 *                       -- Objective: write load() & save()          *
 *                    -- a.2 complete                                 *
 *          31 Dec 98 -- Version a.2.1 begun                          *
 *                       -- Objective:  rename nodeinfo.heapsize to   *
 *                          nodeinfo.listsize                         *
 *                    -- a.2.1 complete                               *
 *                    -- nodeinfo.h 0.1                               *
 *                    -- NodeMetric 0.1.2                             *
 *           1 Jan 99 -- NodeMetric 0.2                               *
 *                                                                    *
 **********************************************************************/


#define filename1 "ABC_MIPS.dat"
#define filename2 "ABC_FLOPS.dat"


typedef struct {
  double key;     /* This would be either bogomips or mflops*/
  double H_short; /* The values generated by HINT */
  double H_int;
  double H_long;
  double H_float;
  double H_double;
  int listsize;   /* only used in the 0th element of the array */
} nodeinfo;


nodeinfo *load(FILE *);
void save(FILE *,nodeinfo *);


nodeinfo *load(FILE *infile) {
  /*********************************************************************
    Reads the linear list stored in infile.  Actually, it'll read in any
    array, so long as the elements of the array are struct nodeinfo's,
    and the listsize attribute of the first element indicates how many
    more elements there are.
    29 Dec 98 -- prototyped
    30 Dec 98 -- coded
    31 Dec 98 -- tweaked
  *********************************************************************/
  nodeinfo *A,B;
  int i;
  char line[31]="";

  fscanf(infile,"%s",line);
  B.key = atof(line);
  fscanf(infile,"%s",line);
  B.H_short = atof(line);
  fscanf(infile,"%s",line);
  B.H_int = atof(line);
  fscanf(infile,"%s",line);
  B.H_long = atof(line);
  fscanf(infile,"%s",line);
  B.H_float = atof(line);
  fscanf(infile,"%s",line);
  B.H_double = atof(line);
  fscanf(infile,"%s",line);
  B.listsize = atoi(line);

  A = calloc(B.listsize+3,sizeof(nodeinfo));/* Why +3?  Because +1 for the */
  A[0] = B;                                 /* 0th element, and +2 because */
                                            /* buildmap will add up to two */
  for ( i = 1 ; i <= A[0].listsize ; i++ ) {/* elements */
    fscanf(infile,"%s",line);
    A[i].key = atof(line);
    fscanf(infile,"%s",line);
    A[i].H_short = atof(line);
    fscanf(infile,"%s",line);
    A[i].H_int = atof(line);
    fscanf(infile,"%s",line);
    A[i].H_long = atof(line);
    fscanf(infile,"%s",line);
    A[i].H_float = atof(line);
    fscanf(infile,"%s",line);
    A[i].H_double = atof(line);
    fscanf(infile,"%s",line);
    A[i].listsize = atoi(line);
  } /* for i */

  return A;
} /* load() */


void save(FILE *outfile, nodeinfo A[]) {
  /*********************************************************************
    Writes the linear list to outfile.
    29 Dec 98 -- prototyped
    30 Dec 98 -- coded
    31 Dec 98 -- tweaked
  *********************************************************************/
  int i;
  for ( i = 0 ; i <= A[0].listsize ; i++ ) {
    fprintf(outfile,"%e\n",A[i].key);
    fprintf(outfile,"%e\n",A[i].H_short);
    fprintf(outfile,"%e\n",A[i].H_int);
    fprintf(outfile,"%e\n",A[i].H_long);
    fprintf(outfile,"%e\n",A[i].H_float);
    fprintf(outfile,"%e\n",A[i].H_double);
    fprintf(outfile,"%d\n",A[i].listsize);
  } /* for i */
} /* save() */
```

C.6. Makefile

```
########################################################################
#                                                                      #
# PROJECT: Asymmetric Load Balancing on a Heterogeneous                #
#          Cluster of PCs                                              #
#          AFIT/GE/ENG/99M-02                                          #
#                                                                      #
# PACKAGE: NodeMetric                                                  #
#          0.2                                                         #
#                                                                      #
# FILE:    metric/Makefile                                             #
#          0.2                                                         #
#                                                                      #
# AUTHOR:  Capt Christopher A. Bohn                                    #
#                                                                      #
# HISTORY: 29 Nov 98 -- NodeMetric Version 0.1                         #
#          10 Dec 98 -- Version a.1 begun                              #
#                       -- Objective: make all object files            #
#                    -- a.1 complete                                   #
#          11 Dec 98 -- Version a.2 begun                              #
#                       -- Objective: make HINT, too                   #
#                    -- a.2 complete                                   #
#                    -- metric/Makefile Version 0.1                    #
#                    -- NodeMetric 0.1.1                               #
#          29 Dec 98 -- Version 0.2 begun                              #
#                       -- Objective: make buildmap                    #
#                    -- metric/Makefile 0.2                            #
#          31 Dec 98 -- NodeMetric 0.1.2                               #
#           1 Jan 99 -- NodeMetric 0.2                                 #
#                                                                      #
########################################################################

### Begin user configurable options ###

CC          = egcs
LINKER      = egcs
OPTFLAGS    = -O3

HINTCC      = gcc
            # HINT may not like your normal compiler
HINTDIR     = ./hint
            # Location of HINT source files
BINDIR      = /home/cbohn/thesis/NPB-mod2/bin
            # Where your application is located; HINT will be
              # placed /below/ this dir

###  End user configurable options  ###

EXECS       = buildmap
OBJS        = metric.o metricmap.o weighnode.o

default: $(OBJS)

all: $(OBJS) $(EXECS)

buildmap: buildmap.o metric.o
	$(LINKER) $(OPTFLAGS) -o buildmap buildmap.o metric.o -lm

buildmap.o: buildmap.c metric.h nodeinfo.h

metric.o: metric.c

metricmap.o: metricmap.c nodeinfo.h

weighnode.o: weighnode.c metric.h metricmap.h

HINT:
	@ cd $(HINTDIR); make CC=$(HINTCC) CFLAGS=$(OPTFLAGS)
	@- mkdir $(BINDIR)/hint
	@ mv $(HINTDIR)/SHORT    $(BINDIR)/hint
```

```
        @ mv $(HINTDIR)/INT      $(BINDIR)/hint
        @ mv $(HINTDIR)/LONGLONG $(BINDIR)/hint
        @ mv $(HINTDIR)/FLOAT    $(BINDIR)/hint
        @ mv $(HINTDIR)/DOUBLE   $(BINDIR)/hint

help:
        @ echo "Options are:          (default is *.o) (all is buildmap & *.o)"
        @ echo "   metric.o"
        @ echo "   metricmap.o"
        @ echo "   weighnode.o"
        @ echo "   HINT"
        @ echo "   all"
        @ echo "   clean"
        @ echo "   veryclean"

clean:
        @- rm -f core *.o *~

veryclean: clean
        @- rm -f hint/core hint/*.o hint/*~

.c.o:
        $(CC) $(OPTFLAGS) -c $*.c
```

Appendix D: Tabulated Raw Results

This appendix contains the raw data for the charts presented in Chapter IV. Performance figures are in Section D.1 (Table D-1 through Table D-3). Cells in the tables for which no corresponding data was collected are shaded gray.

The sizes of the partitions generated by the different weighting approaches are listed in Section D.2 (Table D-4 through Table D-18). The tables list the width of the column-striped partitions before asymmetric load balancing is introduced. For each of the three weightings (BogoMIPS, Mflops, and QUIPS), the tables include the weight for each node, the "fair share" partition size based on the reported weights before rounding and corrections, and the final tile width for each node.

Finally, Section D.3 (Table D-19) contains performance data collected during development, and not as a part of formal experimentation. It is included because it is relevant to Section 4.6.1.

D.1. Performance

Table D-1. Non-load balanced performance – power-of-two number of processors.

	1 processor				2 processor			4 processor			8 processor	
	1x200	1x333	1x400	1x450	1x200 1x450	1x333 1x450	1x400 1x450	1x200 1x333 1x400 1x450	1x333 2x400 1x450	3x400 1x450	1x200 1x333 5x400 1x450	1x333 6x400 1x450
Unbal Chkbd	0.81 nan	2.96 nan	3.54 nan	3.94 nan	38.42	84.01	112.39	7.27 nan	11.00 nan	13.14 nan	141.94	344.55
		2.96 nan	3.54 nan	3.96 nan				7.28 nan			158.59	346.79
		2.96 nan	3.54 nan	3.96 nan				7.28 nan			160.07	347.13
		2.96 nan	3.54 nan	3.96 nan				7.28 nan			160.34	347.49
		2.96 nan	3.54 nan	3.96 nan				connectn timed out			160.52	348.92
				3.96 nan								
				3.96 nan								
				3.96 nan								
				3.96 nan								
				3.96 nan								
Unbal Row Striped				60.08							163.57	
				60.09							167.49	
				60.13							174.72	
				60.15							174.8	
				60.17							174.88	
Unbal Col Striped		42.7	55.15	59.99	38.37	84.93	101.44	76.14	153.95	170.99	166.75	346.35
		42.7	55.55	60.05	38.5	85.42	112.23	76.63	165.07	171.45	169.71	353.61
		42.7	55.57	60.05	38.52	86.69	112.27	76.87	165.13	171.95	170.05	364.21
		42.71	55.62	60.05	38.54	86.7	112.35	77.96	165.22	172.06	174.86	367.98
		42.71	55.66	60.08	38.76	86.78	112.53	77.97	165.39	172.43	175.15	370.35
				60.12								
				60.12								
				60.12								
				60.13								
				60.13								

Table D-2. Load balanced performance – power-of-two number of processors.

	1 processor				2 processor			4 processor			8 processor	
	1x200	1x333	1x400	1x450	1x200 1x450	1x333 1x450	1x400 1x450	1x200 1x333 1x400 1x450	1x333 2x400 1x450	3x400 1x450	1x200 1x333 5x400 1x450	1x333 6x400 1x450
Equal											1.75	
Weight											1.76	
Rw Str											1.77	
Equal					36.32	63.73	82.29	75.08	159.33	164.88	152.25	307.76
Weight					36.89	68.15	92.09	75.1	159.48	164.88	160.33	312.32
Col					36.94	69.09	92.1	75.34	159.48	165.08	164.67	312.73
Striped					36.95	70.34	92.29	75.45	159.75	165.23	165.32	312.88
					36.98	71.88	92.47	75.99	159.88	165.27	165.7	313.18
Bogo					65.71	80.24	96.27	125.43	156.61	164.08	220.02	294.19
MIPS					65.79	81.09	96.36	125.45	156.7	164.42	267.2	299.93
Col					65.8	84.96	96.47	125.46	156.78	164.82	267.54	300.84
Striped					65.85	84.97	96.52	125.47	156.91	165.26	267.65	300.94
					65.88	84.98	96.55	125.48	165.77	165.59	268.8	301.9
Mflops					70.24	78	96.5	131.83	158.3	165.85	201.72	291.85
Col					71.03	82.38	96.51	131.87	158.3	166.3	240.26	295.46
Striped					71.06	85.01	96.55	131.93	158.37	166.55	264.84	296.21
					71.07	85.15	96.71	132.23	158.41	167.13	265.84	296.27
					71.1	85.17	96.72	134.26	158.45	167.15	265.97	299.38
QUIPS					63.93	69.75	96.97	142.64	159.51	179.07	254.39	307.78
Col					63.93	79.53	97.01	142.66	163.35	179.4	282.83	319.87
Striped					63.94	82.14	97.02	142.83	164.29	179.41	284.28	329.99
					63.95	86.4	97.05	142.9	164.73	180.23	284.78	330.13
					63.95	86.43	97.11	142.91	164.74	181.36	285.57	331.05

Table D-3. Load balanced & non-load balanced performance – non-power-of-two number of processors.

	3 processor	7 processor	9 processor	10 processor	11 processor		12 processor
	1x333 1x400 1x450	1x333 5x400 1x450	1x200 1x333 5x400 1x450	2x333 6x400 1x450	1x200 3x333 6x400 1x450	4x333 6x400 1x450	1x200 4x333 6x400 1x450
Unbal	121.09	293.64	122.26	96.39	123.11	117.86	259.99
Col	121.15	297.1	188.81	119.3	124.11	120.15	264.3
Striped	121.16	298.28	278.96	167.37	213.49	122.35	269.26
	121.22	299.87	290.61	318.3	230.92	132.77	275.63
	121.22	301.34	317.44	347.32	257.27	211.18	280.43
Equal	126.26	269.21	198.47	105.95	118.71	109.91	77.01
Weight	126.26	272.27	243.63	162.19	131.16	118.09	78.24
Col	126.38	274.36	255.87	196.32	131.27	127.75	102.81
Striped	126.4	275.12	256.79	254.03	168.28	135.1	108.06
	126.61	276.87	280.72	295.09	175.44	273.85	109.94
Bogo	126.7	248.07	235.69	134.41	120.64	59.74	127.5
MIPS	126.86	271.21	256.91	264.25	133.08	99.7	168.87
Col	126.92	271.23	308.09	264.69	166.42	141.23	171.7
Striped	126.95	271.95	317.38	276.99	204.18	182.98	176.21
	126.98	272.5	321.39	298.71	211.22	341.76	235.77
Mflops	125.76	256.26	217.29	146.76	158.83	85.8	127.37
Col	126.09	266.82	229.28	149.48	204.25	252.25	147.61
Striped	126.37	270.84	258.54	151.42	216.18	304.83	156.58
	126.4	271.34	273.35	277.54	223.76	334.77	159.17
	127.24	272.29	274.17	334.84	318.17	344.89	229.04
QUIPS	133.31	275.26	232.54	131.95	116.04	85.21	102.42
Col	133.84	289.51	311.23	135.6	122.78	133.81	145.63
Striped	134.31	291.77	331.31	136.58	125.8	276.51	152.31
	134.53	292.31	331.55	206.91	178.6	340.42	169.05
	134.86	295.13	347.72	251.68	200.34	342.54	267.83

D.2. Partitioning

Table D-4. Two-processor partitioning (1x200 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC11	32	445.64	54.29358	54	71.9	49.7309	50	16148280	49.95508	50
ABC12	32	79.67	9.706421	10	20.63	14.2691	14	4540104	14.04492	14

Table D-5. Two-processor partitioning (1x333 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	32	332.6	27.35197	27	54.71	27.65532	28	12153570	27.48331	27
ABC11	32	445.64	36.64803	37	71.9	36.34468	36	16148280	36.51669	37

Table D-6. Two-processor partitioning (1x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC09	32	396.49	30.13235	30	64.53	29.80816	30	14875540	30.68721	31
ABC11	32	445.64	33.86765	34	74.02	34.19184	34	16148280	33.31279	33

Table D-7. Three-processor partitioning (1x333 1x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	22	332.6	18.12025	18	53.54	18.03737	18	12153570	18.01472	18
ABC09	21	396.49	21.60101	22	64.53	21.73985	22	14875540	22.04938	22
ABC11	21	445.64	24.27874	24	71.9	24.22277	24	16148280	23.93591	24

Table D-8. Four-processor partitioning (1x200 1x333 1x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	16	332.6	16.96939	17	53.54	16.42567	16	12153570	16.3007	16
ABC09	16	396.49	20.22908	20	64.53	19.79733	20	14875540	19.95148	20
ABC11	16	445.64	22.73673	23	69.91	21.44787	22	16148280	21.65851	22
ABC12	16	79.67	4.064796	4	20.63	6.329131	6	4540104	6.089311	6

Table D-9. Four-processor partitioning (1x333 2x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	16	332.6	13.54769	14	55.92	14.04088	14	12153570	13.39861	13
ABC06	16	396.49	16.1501	16	64.53	16.20275	16	14875540	16.39942	16
ABC09	16	396.49	16.1501	16	64.53	16.20275	16	14875540	16.39942	16
ABC11	16	445.64	18.15211	18	69.91	17.55361	18	16148280	17.80255	19

Table D-10. Four-processor partitioning (3x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC06	16	396.49	15.51905	15	64.53	15.77028	16	14875540	15.66493	15
ABC07	16	396.49	15.51905	16	64.53	15.77028	16	14875540	15.66493	16
ABC08	16	396.49	15.51905	16	62.91	15.37437	15	14875540	15.66493	16
ABC11	16	445.64	17.44284	17	69.91	17.08508	17	16148280	17.00521	17

Table D-11. Seven-processor partitioning (1x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	10	332.6	7.710536	8	53.54	7.702731	8	12153570	7.575301	8
ABC05	9	396.49	9.191673	9	62.91	9.050781	9	14875540	9.2719	9
ABC06	9	396.49	9.191673	9	64.53	9.283848	9	14875540	9.2719	9
ABC07	9	396.49	9.191673	9	64.53	9.283848	9	14875540	9.2719	9
ABC08	9	396.49	9.191673	9	62.91	9.050781	9	14875540	9.2719	9
ABC09	9	396.49	9.191673	9	64.53	9.283848	9	14875540	9.2719	9
ABC11	9	445.64	10.3311	11	71.9	10.34416	11	16148280	10.0652	11

Table D-12. Eight-processor partitioning (1x200 1x333 5x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	8	332.6	7.494261	5	53.54	7.313115	5	12153570	7.254533	5
ABC05	8	396.49	8.933853	9	64.53	8.814257	9	14875540	8.879291	9
ABC06	8	396.49	8.933853	9	64.53	8.814257	9	14875540	8.879291	9
ABC07	8	396.49	8.933853	9	64.53	8.814257	9	14875540	8.879291	9
ABC08	8	396.49	8.933853	9	64.53	8.814257	9	14875540	8.879291	9
ABC09	8	396.49	8.933853	9	64.53	8.814257	9	14875540	8.879291	9
ABC11	8	445.64	10.04132	10	71.9	9.820937	10	16148280	9.638997	10
ABC12	8	79.67	1.795153	4	20.46	2.794664	4	4540104	2.710013	4

Table D-13. Eight-processor partitioning (1x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC03	8	332.6	6.742219	7	53.54	6.705597	7	12153570	6.616715	7
ABC05	8	396.49	8.03735	8	64.53	8.082035	8	14875540	8.098625	8
ABC06	8	396.49	8.03735	8	64.53	8.082035	8	14875540	8.098625	8
ABC07	8	396.49	8.03735	8	64.53	8.082035	8	14875540	8.098625	8
ABC08	8	396.49	8.03735	8	62.91	7.879139	8	14875540	8.098625	8
ABC09	8	396.49	8.03735	8	64.53	8.082035	8	14875540	8.098625	8
ABC10	8	396.49	8.03735	8	64.53	8.082035	8	14875540	8.098625	8
ABC11	8	445.64	9.033682	9	71.9	9.005088	9	16148280	8.791537	9

Table D-14. Nine-processor partitioning (2x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC02	8	332.6	6.09964	6	53.54	6.078479	6	12153570	5.996735	6
ABC03	7	332.6	6.09964	6	54.71	6.211311	6	12153570	5.996735	6
ABC05	7	396.49	7.271335	7	64.53	7.32619	7	14875540	7.339792	7
ABC06	7	396.49	7.271335	7	62.91	7.142269	7	14875540	7.339792	7
ABC07	7	396.49	7.271335	7	64.53	7.32619	7	14875540	7.339792	7
ABC08	7	396.49	7.271335	7	64.53	7.32619	7	14875540	7.339792	7
ABC09	7	396.49	7.271335	7	64.53	7.32619	7	14875540	7.339792	7
ABC10	7	396.49	7.271335	8	64.53	7.32619	8	14875540	7.339792	8
ABC11	7	445.64	8.17271	9	69.91	7.93699	9	16148280	7.967779	9

Table D-15. Ten-processor partitioning (3x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC01	7	332.6	5.568886	5	54.71	5.68748	5	12153570	5.482985	5
ABC02	7	332.6	5.568886	5	53.54	5.56585	5	12153570	5.482985	5
ABC03	7	332.6	5.568886	5	53.54	5.56585	5	12153570	5.482985	5
ABC05	7	396.49	6.638628	7	62.91	6.539926	7	14875540	6.71098	7
ABC06	6	396.49	6.638628	7	64.53	6.708336	7	14875540	6.71098	7
ABC07	6	396.49	6.638628	7	64.53	6.708336	7	14875540	6.71098	7
ABC08	6	396.49	6.638628	7	62.91	6.539926	7	14875540	6.71098	7
ABC09	6	396.49	6.638628	7	64.53	6.708336	7	14875540	6.71098	7
ABC10	6	396.49	6.638628	7	64.53	6.708336	7	14875540	6.71098	7
ABC11	6	445.64	7.461571	7	69.91	7.267624	7	16148280	7.285166	7

Table D-16. Eleven-processor partitioning (1x200 3x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC01	6	332.6	5.455184	4	54.71	5.465193	4	12153570	5.312951	4
ABC02	6	332.6	5.455184	4	54.71	5.465193	4	12153570	5.312951	4
ABC03	6	332.6	5.455184	4	53.54	5.348317	5	12153570	5.312951	4
ABC05	6	396.49	6.503084	6	64.53	6.446151	6	14875540	6.502865	6
ABC06	6	396.49	6.503084	7	64.53	6.446151	6	14875540	6.502865	7
ABC07	6	396.49	6.503084	7	64.53	6.446151	6	14875540	6.502865	7
ABC08	6	396.49	6.503084	7	64.53	6.446151	7	14875540	6.502865	7
ABC09	6	396.49	6.503084	7	64.53	6.446151	7	14875540	6.502865	7
ABC10	6	396.49	6.503084	7	64.53	6.446151	7	14875540	6.502865	7
ABC11	5	445.64	7.309225	7	69.91	6.98358	8	16148280	7.059245	7
ABC12	5	79.67	1.306718	4	20.63	2.06081	4	4540104	1.984713	4

Table D-17. Eleven-processor partitioning (4x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC01	6	332.6	5.123105	5	53.54	5.117095	5	12153570	5.050316	5
ABC02	6	332.6	5.123105	5	53.54	5.117095	5	12153570	5.050316	5
ABC03	6	332.6	5.123105	5	53.54	5.117095	5	12153570	5.050316	5
ABC04	6	332.6	5.123105	5	53.54	5.117095	5	12153570	5.050316	5
ABC05	6	396.49	6.107216	6	64.53	6.167466	6	14875540	6.181408	6
ABC06	6	396.49	6.107216	6	62.91	6.012634	6	14875540	6.181408	6
ABC07	6	396.49	6.107216	6	64.53	6.167466	6	14875540	6.181408	6
ABC08	6	396.49	6.107216	6	64.53	6.167466	6	14875540	6.181408	6
ABC09	6	396.49	6.107216	6	64.53	6.167466	6	14875540	6.181408	6
ABC10	5	396.49	6.107216	6	64.53	6.167466	6	14875540	6.181408	6
ABC11	5	445.64	6.864283	8	69.91	6.68166	8	16148280	6.710285	8

Table D-18. Twelve-processor partitioning (1x200 4x333 6x400 1x450).

	Original	BogoMIPS			Mflops			QUIPS		
	Partition	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n	Weight	Suggest'n	Part'n
ABC01	6	332.6	5.02672	4	53.54	4.963727	4	12153570	4.905705	4
ABC02	6	332.6	5.02672	4	53.54	4.963727	4	12153570	4.905705	4
ABC03	6	332.6	5.02672	4	54.71	5.072198	5	12153570	4.905705	4
ABC04	6	332.6	5.02672	5	52.43	4.860818	5	12153570	4.905705	5
ABC05	5	396.49	5.992316	6	64.53	5.982617	6	14875540	6.004409	6
ABC06	5	396.49	5.992316	6	62.91	5.832426	6	14875540	6.004409	6
ABC07	5	396.49	5.992316	6	64.53	5.982617	6	14875540	6.004409	6
ABC08	5	396.49	5.992316	6	64.53	5.982617	6	14875540	6.004409	6
ABC09	5	396.49	5.992316	6	64.53	5.982617	6	14875540	6.004409	6
ABC10	5	396.49	5.992316	6	64.53	5.982617	6	14875540	6.004409	6
ABC11	5	445.64	6.73514	7	69.91	6.4814	6	16148280	6.518142	7
ABC12	5	79.67	1.204085	4	20.63	1.91262	4	4540104	1.832582	4

D.3. Additional Results Obtained During Development

Table D-19. Results collected during development, using NPB-serial, hub, and switch. All processors are 400 MHz Pentium IIs.

	1 processor			2 processor		4 processor	
	NBP-serial	hub	switch	hub	switch	hub	switch
Unbal Chkbd	38.71	49.06	48.97	96.88	96.09	176.92	181.84
	38.73	49.17	49.16	96.99	96.10	178.67	181.87
	38.78	49.21	49.19	97.05	96.11	178.68	181.87
Unbal Row Striped			48.82		95.38		172.85
			48.95		95.39		173.14
			49.00		95.40		173.17
Unbal Col Striped			48.77		96.12		171.21
			48.78		96.13		171.79
			48.79		96.14		173.14

References

[1] Adamson, Iain T. *Data Structures and Algorithms: A First Course.* London: Springer, 1996.

[2] Aeronautical Systems Center. “CFD Overview,” updated 18 November 1998. http://www.asc.hpc.mil/PET/CFD/overview.html .

[3] Aeronautical Systems Center. “CFD-04; Grid Generation Support for Grand Challenge Problems,” updated 19 November 1998. http://www.asc.hpc.mil/PET/CFD/Projects/cfd04.html .

[4] Aeronautical Systems Center. “Introduction to Parallel Programming,” updated 20 August 1997. http://www.asc.hpc.mil/webtrn/p11_18/html/parallel-intro/ParallelIntro.html .

[5] Aeronautical Systems Center. “Introduction to Parallel Programming: The Need for Faster Machines,” updated 20 August 1997. http://www.asc.hpc.mil/webtrn/p11_18/html/parallel-intro/ParallelIntro.html#needf orfastermachines .

[6] Alta Technology. “Alta Technology Menu,” http://www.altatech.com/ , printed 28 January 1999.

[7] Anderson, John D., Jr. *Computational Fluid Dynamics: The Basics with Applications.* New York: McGraw-Hill, 1995.

[8] Anderson, Thomas E., David E. Culler, David A. Patterson, and the NOW Team. “A Case for NOW (Networks of Workstations),” *IEEE Micro* 15:1 (January 1995), pp54-64.

[9] Argonne National Laboratory. “MPICH-A Portable Implementation of MPI,” http://www.mcs.anl.gov/Projects/mpi/mpich/index.html , printed 18 August 1998.

[10] Bailey, David H., Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. “The NAS Parallel Benchmarks 2.0,” December 1995. NAS-95-020. http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-95-011/NAS-95-020.ps .

[11] Barszcz, E., R. Fatoohi, V. Venkatkrishnan, and S. Weeratunga. "Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors," April 1993. RNR-93-007. http://science.nas.nasa.gov/Pubs/TechReports/RNRreports/ebarszcz/RNR-93-007/RNR-93-007.ps .

[12] Bell, Gordon. "The Future of High Performance Computers in Science and Engineering," *Communications of the ACM* 32:9 (September 1989), pp1091-1101.

[13] Beowulf Mailing List Archive http://www.beowulf.org/listarchives/beowulf/ .

[14] Brey, Barry B. *The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1997.

[15] Chandy, K. Mani, and Jayadev Misra. *Parallel Programming Design: A Foundation*. Reading, MA: Addison-Wesley, 1987.

[16] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 1990.

[17] Corradi, Antonio, Letizia Leonardi, and Franco Zambonelli. "Diffusive Load-Balancing Policies for Dynamic Applications," *IEEE Concurrency* 7:1 (January-March 1999), pp22-31.

[18] Cox, Alan. "UK.LINUX.ORG," printed 5 February 1999. http://www.uk.linux.org/ .

[19] Cygnus Solutions. "egcs Project Home Page," updated 15 January 1999. http://egcs.cygnus.com/ .

[20] Davis, Stephen R. *C++ for Dummies*, 2d ed. Foster City, CA: IDG Books Worldwide, Inc., 1996.

[21] DCG Computers, Inc. "Do you want to compute faster, DCG Inc can show you the way," http://www.dcginc.com/ , updated 12 January 1999.

[22] Decker, Thomas, Reinhard Lüling, and Stefan Tschöke. "A Distributed Load Balancing Algorithm for Heterogeneous Parallel Computing Systems," *Proceedings, International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998, pp951-957.

[23] Department of Energy. "The Accelerated Strategic Computing Initiative (ASCI)," http://www.asci.doe.gov/ , printed 9 February 1999.

[24] Dietz, Hank. "Linux Parallel Processing Sites," updated 23 March 1998, http://yara.ecn.purdue.edu/~pplinux/Sites/ .

[25] Dongarra, Jack J., Hans W. Meuer, and Erich Strohmaier. "TOP500 Supercomputer Sites," 11th ed. (18 June 1988), http://www.top500.org/lists/1998/06/top500_9806.ps.gz .

[26] Dongarra, Jack J., Hans W. Meuer, and Erich Strohmaier. "TOP500 Supercomputer Sites," 12th ed. (5 November 1998), http://www.top500.org/lists/1998/11/top500_9811.ps.gz .

[27] Dowd, Kevin, and Charles Severance. *High Performance Computing*, 2d ed. Cambridge, MA: O'Reilly and Associates, 1998.

[28] Dubrovsky, Alexander, Roy Friedman, and Assaf Schuster. "Load Balancing in a Distributed Shared Memory System," *International Journal of Applied Software Techniques*, vol. 3 (March 1998), pp167-202, http://www.cs.technion.ac.il/~assaf/publications/alex.doc .

[29] Ellis, T.M.R., Ivor R. Philips, and Thomas M. Lahey. *Fortran 90 Programming.* Harlow, England: Addison-Wesley, 1994.

[30] Gilly, Daniel, and the Staff of O'Reilly & Associates, Inc. *UNIX in a Nutshell: System V Edition.* Cambridge: O'Reilly & Associates, 1992.

[31] Gindhart, David C. *A Comparative Analysis of Networks of Workstations and Massively Parallel Processors for Signal Processing.* MSCE thesis, AFIT/GCE/ENG/97D-01. Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, December 1997.

[32] Grassmann, Winfried Karl, and Jean-Paul Tremblay. *Logic and Discrete Mathematics: A Computer Science Perspective.* Upper Saddle River, NJ: Prentice Hall, 1996.

[33] Greenfield, Larry. *The Linux User's Guide*, Beta-1. [51], 1996.

[34] Gustafson, John L., and Quinn O. Snell, "HINT: A New Way to Measure Computer Performance," updated 21 Oct 97, http://www.scl.ameslab.gov/Publications/HINT/ComputerPerformance.html , printed 26 Aug 98.

[35] Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*, 2d ed. San Francisco: Morgan Kaufmann Publishers, 1996.

[36] High Performance Computing Modernization Program Office. "Computational Fluid Dynamics(CFD)," updated 8 January 1999. http://www.hpcmo.hpc.mil/Htdocs/CTAs/cfd.html .

[37] High Performance Computing Modernization Program Office. "Computational Electromagnetics and Acoustics(CEA)," updated 30 September 1998. http://www.hpcmo.hpc.mil/Htdocs/CTAs/cea.html .

[38] High Performance Computing Modernization Program Office. "HPCMP Computational Technology Areas," printed 17 January 1999. http://www.hpcmo.hpc.mil/Htdocs/CTAs/CTAs.html .

[39] "HINT (Hierarchical INTegration)," ftp://ftp.scl.ameslab.gov/pub/hint .

[40] Howe, Denis. "The Free On-Line Dictionary of Computing," 27 October 98, http://wombat.doc.ic.ac.uk/ .

[41] HPCwire. "HPC Community Honors its Best, Brightest, & Fastest," *HPCwire* (18 December 1998) article 14485.

[42] Hughes, William F., and John A. Brighton. *Schaum's Outline of Theory and Problems of Fluid Dynamics*, 2d ed. New York: McGraw-Hill, 1991.

[43] Hwang, Kai and Zhiwei Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. Boston: McGraw-Hill, 1997.

[44] Ince, D.C. *An Introduction to Discrete Mathematics, Formal System Specification, and Z*, 2d ed. Oxford: Clarendon Press, 1992.

[45] Intel Corporation. *Intel Express 510T Switch User Guide*. Hillsboro, OR: Intel Corporation, 1997.

[46] Kelley, Al, and Ira Pohl. *A Book on C: Programming in C*, 2d ed. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1990.

[47] Kirsch, Christian, Florain Lohoff, and André von Raison. "CLOWN - Cluster of Working Nodes," *iX* (January 1999). http://www.heise.de/ix/artikel/E/1999/01/010/ .

[48] Kumar, Vipin, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City CA: The Benjamin/Cummings Publishing Company, 1994.

[49] Latin, Robert M. *The Influence of Surface Roughness on Supersonic High Reynolds Number Turbulent Boundary Layer Flow.* PhD dissertation, AFIT/DS/ENY/98M-02. Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, September 1998.

[50] Ligon, Walter B. III. "BEOWULF Underground FAQ – What IS a BEOWULF parallel processing computer?" printed 20 January 1999. http://webwulf.parl.clemson.edu/underground/faq.html#whatis .

[51] The Linux Documentation Project (various authors), http://MetaLab.unc.edu/LDP/ldp.html .

[52] Merkey, Phil. "The Beowulf Consortium," updated 7 October 1998, http://www.beowulf.org/consortium.html .

[53] Merkey, Phil. "Beowulf Project at CESDIS," updated 11 September 1998, http://www.beowulf.org/ .

[54] Merkey, Phil. "Beowulf Software," updated 20 January 1999. http://www.beowulf.org/software/software.html .

[55] Merkey, Phil. "Re: Number of subscribers to Beowulf Mailing List." Electronic mail to author, 4 February 1999.

[56] Microsoft Corporation. "Microsoft Windows," updated 15 January 1999. http://www.microsoft.com/windows/ .

[57] Milton, J.S., and Jesse C. Arnold. *Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences*, 3d ed. New York: McGraw-Hill, 1995.

[58] MindShare Inc. and Tom Shanley. *Pentium Pro and Pentium II System Architecture*, 2d ed. Reading MA: Addison-Wesley, 1998.

[59] Myricom. "Myrinet Products Index," updated 17 April 1998. http://www.myri.com/myrinet/ .

[60] National Science Foundation. "NSF Report of the Grand Challenges, National Challenges, and Multidisciplinary Challenges Grantees Works," updated 13 January 1998. http://www.cise.nsf.gov/general/challenge/execsummary.html .

[61] NAS Parallel Benchmarks 2.3. Author's examination of source code downloaded from [99].

[62] Netlib. "Benchmark Programs and Reports," printed 1 February 1999. http://www.netlib.org/benchmark/ .

[63] Paralogic, Inc. "Paralogic, Inc.," http://www.plogic.com/ , printed 28 January 1999. See also Paralogic, Inc. "xtreme MACHINES," http://www.xtreme-machines.com/ , printed 28 January 1999.

[64] Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* San Francisco: Morgan Kaufmann Publishers, 1994.

[65] Pentium Compiler Group. "Pentium Compiler Group," updated 12 December 1998. http://www.goof.com/pcg/ .

[66] Radajewski, Jacek, and Douglas Eadline. "Beowulf HOWTO" v1.1.1 (22 November 1998), in [51].

[67] Raymond, Eric S. "The Jargon File," v4.0.0 (25 June 1996), http://www.tuxedo.org/~esr/jargon/ . Corresponds to Raymond, Eric S. *The New Hacker's Dictionary*, 3d ed. Cambridge, MA: The MIT Press, 1996.

[68] Raymond, Eric S. "The Cathedral and the Bazaar," v1.40 (11 August 1998), http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.ps .

[69] Raymond, Eric S. "Homesteading the Noosphere," v1.10 (11 July 1998), http://www.tuxedo.org/~esr/writings/homesteading/homesteading.ps .

[70] Red Hat Software, Inc. "What is Linux?" printed 20 January 1999. http://www.redhat.com/linux_what.phtml .

[71] Reggiani, Luca. "PC Clusters," updated April 1998, http://www.ce.unipr.it/~lucareg/ .

[72] Ridge, Daniel, Donald Becker, Phillip Merkey, and Thomas Sterling. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," *Proceedings, IEEE Aerospace*, 1997. http://cesdis.gsfc.nasa.gov/beowulf/papers/aa97/final.ps . (Presentation Slides) http://www.beowulf.org/slides/index.html .

[73] Robbins, Carla Anne. "New High-Speed PCs Raise Risk of Nuclear Proliferation," *iX* (14 December 1998). http://www.wsj.com/public/current/articles/SB13589580119560500.htm .

[74] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall, 1991.

[75] Saini, Subhash, and David H. Bailey. "NAS Parallel Benchmarks Results 3-95," April 1995. NAS-95-011. http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-95-011/NAS-95-011.ps .

[76] Salamon, Wayne. "TCP Performance Drop in Linux," printed 11 February 1999. http://www.multikron.nist.gov/scalable/misc_info/Linux_TCP.html .

[77] Salmon, John, Christopher Stein, and Thomas Sterling. "Scaling of Beowulf-class Distributed Systems," *Proceedings, IEEE/ACM Supercomputing* (CD-ROM), 1998, \sc98\TechPapers\sc98_FullAbstracts\Salmon793\v7.htm .

[78] Saphir, William, Alex Woo, and Maurice Yarrow. "NAS Parallel Benchmarks 2.1 Results," August 1996. NAS-96-010. http://science.nas.nasa.gov/Software/NPB/Reports/NAS-96-010/NAS-96-010.ps .

[79] Sharifi, Mohsen, and Kamran Karimi "DIPC: The Linux Way of Distributed Programming," *Linux Journal* 57 (January 1999), pp10-17.

[80] Silicon Graphics, Inc. "Silicon Graphics – Cray T3E Datasheets: Cray T3E-1200," printed 25 February 1999, http://www.sgi.com/t3e/t3e_1200.html .

[81] Silva, Luis Moura. "Number-Crunching with Java Applications," *Proceedings, International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998, pp379-385.

[82] Snell, Quinn, Glenn Judd, and Mark Clement. "Load Balancing in a Heterogeneous Supercomputing Environment," *Proceedings, International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998, pp951-957.

[83] The Standard Performance Evaluation Corporation. "Welcome To SPEC," updated 21 August 1998, http://www.spec.org/ .

[84] Sterling, Thomas, Donald J. Becker, John E. Dorband, Daniel Savarese, Udaya A. Ranawake, and Charles V. Packer. "Beowulf: A Parallel Workstation for Scientific Computing," *Proceedings, International Conference on Parallel Processing*, 1995. http://www.beowulf.org/papers/ippc95/final.ps .

[85] Sterling, Thomas, Tom Cwik, Don Becker, John Salmon, Mike Warren, and Bill Nitzberg. "An Assessment of Beowulf-class Computing for NASA Requirements: Initial Findings from the First NASA Workshop on Beowulf-class Clustered Computing," *Proceedings, IEEE Aerospace Conference* , 1998. http://loki-www.lanl.gov/papers/p312.ps .

[86] Tanenbaum, Andrew S. *Distributed Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1995.

[87] Tanaka, Y., M. Matsuda, K. Kubota, and M. Sato. "Performance Improvement by Overlapping Computation and Communication on SMP Clusters," *Proceedings, International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998, pp275-282.

[88] Tweten, Dave. "Whitney Project Home Page," updated 10 December 1998. http://parallel.nas.nasa.gov/Parallel/Projects/Whitney/ .

[89] University of Mannheim. "TOP500 Supercomputing Sites," updated 19 November 1998. http://www.top500.org/ .

[90] Valloppillil, Vinod. "Open Source Software: A (New?) Development Methodology," v1.00 (11 August 1998), with annotations by Eric S. Raymond. aka "The Halloween Document," http://www.opensource.org/halloween1.html .

[91] Van der Linden, Peter. *Just Java.* Mountain View, CA: SunSoft Press, 1996.

[92] Van Dorst, Wim. "BogoMips mini-HOWTO" (13 January 1999), in [51].

[93] Wall, Larry. `patch(1)` man page.

[94] Warren, Michael S., Donald J. Becker, M. Patrick Goda, John K. Salmon, and Thomas Sterling. "Parallel Supercomputing with Commodity Components," *Proceedings, International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997. http://loki-www.lanl.gov/papers/pdpta97/ .

[95] Warren, Michael S., John K. Salmon, Donald J. Becker, M. Patrick Goda, Thomas Sterling, and Grégoire S. Winckelmans. "Pentium Pro Inside: I. A Treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac," *Proceedings, IEEE/ACM Supercomputing*, 1997, http://loki-www.lanl.gov/papers/sc97 .

[96] Warren, Michael S., Timothy C. Germann, Peter S. Lomdahl, David M. Beazley, and John K. Salmon. "Avalon: An Alpha/Linux Cluster Achieves 10 Gflops for $150k," *Proceedings, IEEE/ACM Supercomputing* (CD-ROM), 1998, \sc98\TechPapers\sc98_FullAbstracts\Warren1549\index.htm .

[97] Warren, Michael S., and M. Patrick Goda. "Loki – Commodity Parallel Processing," updated 17 April 1998. http://loki-www.lanl.gov/ .

[98] Wirzenius, Lars. *The Linux System Administrators' Guide*, 0.6. [51], 1997.

[99] Woo, Alex. “The NAS Parallel Benchmarks,” updated 1 August 1997. http://science.nas.nasa.gov/Software/NPB/ .

[100] Yarrow, Maurice. Telephone interview with author, 22 January 1999.

[101] Yarrow, Maurice, and Rob Van der Wijngaart. “Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes,” November 1997. NAS-97-032. http://science.nas.nasa.gov/Software/NPB/Reports/NAS-97-032/NAS-97-032.ps .

[102] Yates, Dustin E. *Modeling and Simulation Support for Parallel Algorithms in a High-Speed Network.* MSCE thesis, AFIT/GCS/ENG/97D-20. Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH, December 1997.

Vita

Captain Christopher Alan Bohn was born on June 7, 1970 in Fort Walton Beach, Florida. After graduating from Desert High School, Edwards, California in 1988, he attended Purdue University, where he received a Bachelor of Science in Electrical Engineering. Upon graduation, he was commissioned a second lieutenant in the United States Air Force on December 21, 1992.

His first assignment was to Minot Air Force Base, North Dakota, where he served in various positions, culminating as Chief, Command Post Operations, Fifth Bomb Wing. While stationed at Minot AFB, Captain Bohn earned a Master of Science from the University of North Dakota, in Space Studies. Following his permanent change of station in August 1997, he enrolled in the Air Force Institute of Technology.

Permanent Address: 16454 Green Pines Dr
Ballwin MO 63011

cbohn@computer.org