The University of Maine
DigitalCommons@UMaine

Electronic Theses and Dissertations | Fogler Library

2004

The Effects of Microprocessor Architecture on Speedup in Distrbuted Memory Supercomputers

Glen L. Beane

Follow this and additional works at: http://digitalcommons.library.umaine.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Beane, Glen L., "The Effects of Microprocessor Architecture on Speedup in Distrbuted Memory Supercomputers" (2004). *Electronic Theses and Dissertations*. 217.
http://digitalcommons.library.umaine.edu/etd/217

This Open-Access Thesis is brought to you for free and open access by DigitalCommons@UMaine. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of DigitalCommons@UMaine.

THE EFFECTS OF MICROPROCESSOR ARCHITECTURE ON SPEEDUP IN DISTRBUTED MEMORY SUPERCOMPUTERS

By

Glen L. Beane

B.S. University of Maine, 2002

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Science)

The Graduate School

The University of Maine

August, 2004

Advisory Committee:

George Markowsky, Professor and Chair of Computer Science, Advisor

Bruce Segee, Associate Professor of Electrical and Computer Engineering

Thomas Wheeler, Assistant Professor of Computer Science

THE EFFECTS OF MICROPROCESSOR ARCHITECTURE ON SPEEDUP IN DISTRBUTED MEMORY SUPERCOMPUTERS

By Glen L. Beane

Thesis Advisor: Dr. George Markowsky

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
August, 2004

Amdahl's Law states that speedup in moving from one processor to N identical processors can never be greater than N, and in fact usually is lower than N because of operations that must be done sequentially. Amdahl's Law gives us the following formula for speedup:

$$Speedup \leq \frac{S + P}{S + P/N}$$

where N is the number of processors, S is the percentage of the code that is serial (i.e., cannot be parallelized), and P is the percentage of code that is parallelizable. We can substitute $1 - S$ for P in the above formula and we see that as S approaches zero speedup approaches N. It can also be shown that seemingly small values of S can severely limit the maximum speedup.

Researchers at the University of Maine saw speedups that seemed to contradict Amdahl's Law, and identified an assumption made by the law that is not always true. When this assumption is not true, it is possible to achieve speedups that are larger than the theoretical maximum speedup of N given by Amdahl's Law.

The assumption in question is that the computer performance scales linearly as the size of the problem is reduced by dividing it over a larger number of processors. This assumption is not valid for computers with tiered memory.

In this thesis we investigate superlinear speedup through a series of test programs specifically designed to exhibit superlinear speedup. After demonstrating these programs show superlinear speedup, we suggest methods for detecting the potential for superlinear speedup in a variety of algorithms.

ACKNOWLEDGMENTS

This work has been supported in part by the US Army Space and Missile Defense Command, contract number DASG60-02-C-0086. The content of this work does not necessarily reflect the position nor the policy of the U.S. government, and no official endorsement should be inferred.

I would like to thank all of the professors in the Department of Computer Science that I've had the opportunity to learn from over the years at the University of Maine. I'd also like to give special thanks to Dr. Laurence Latour, my undergraduate academic advisor.

I'd like to thank the members of my thesis committee: Dr. George Markowsky, Dr. Thomas Wheeler, and especially Dr. Bruce Segee whom with I've worked very closely on this research. I would also like to thank Caleb Carter, who worked with Dr. Segee on some early research that helped to inspired this thesis.

I also would not be in this position today if it were not for everyone involved in the Supercluster Distributed Memory Technology Research Group at the University of Maine. If it were not for this group I would not have had the opportunity to learn so much about cluster computing. While I can't thank everyone individually, I feel the need to single out Eric Wages since he was key in bringing me into the project.

Lastly I would be remiss if I did not thank my family. Thanks to my sister Rebecca Ambrose and my brother-in-law Thomas Ambrose for their support. Very special thanks to my parents, Donald and Camille Beane, who supported me throughout my educational career. Their support has meant the world to me and I can never thank them enough. I don't know if I could have done it without them. Another person that deserves special thanks is my fiancee, Alison Charloux, who was very understanding when I needed to sit at my computer for hours at a time "working on my thesis", or when I got "stressed out".

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

Chapter

1 Introduction 1
  1.1 Growing Computational Power 1
  1.2 Growing Demand for Computational Power 3
  1.3 Parallel Computing Overview 4
  1.4 Our Work 6

2 Significant Prior Research 7
  2.1 Amdahl's Law 7
  2.2 Gustafson's Scaled Speedup 9
    2.2.1 Scaled Speedup Explained 12
    2.2.2 Fixed Time vs Fixed Size 13
  2.3 Parallel Overhead 13
  2.4 Superlinear Speedup 14
  2.5 Chapter Conclusions 18

3 Research Questions 20
  3.1 Problem Introduction 20
  3.2 Our Hypothesis 22
    3.2.1 Amdahl's Assumption 22
    3.2.2 Problem with this Assumption 22
  3.3 Chapter Conclusions 23

4 Research Methods 25
  4.1 Experiment Overview 25
  4.2 Test Programs 25
    4.2.1 Program 1 26
    4.2.2 Program 2 27
    4.2.3 Program 3 28
    4.2.4 Data Set and Number of Iterations 29
  4.3 Test Procedure 30
  4.4 Additional Minor Experiments 30
    4.4.1 Communication Interval Tests 31
    4.4.2 Multiple Array Tests 31

4.4.2.1 MultiArrayTest 1 Pseudo Code 31
4.4.2.2 MultiArrayTest 2 Pseudo Code 32
4.5 Chapter Summary 33

5 Research Results 34
5.1 Introduction 34
5.2 Results, Two Processors Per Node 34
5.2.1 Program 1 Results 34
5.2.2 Program 2 Results 34
5.2.3 Program 3 Results 36
5.3 Results, One Processor Per Node 36
5.4 Chapter Conclusions 37

6 Discussion 38
6.1 Program 1 Discussion 38
6.1.1 Two Processors Per Node 38
6.1.2 One Processor Per Node 39
6.2 Program 2 Discussion 39
6.3 Program 3 Discussion 40
6.3.1 Communication Interval Tests 40
6.3.2 Communication Interval Test Discussion 42
6.4 Multiple Array Tests 44
6.5 Chapter Summary 44

7 Summary and Conclusion 47
7.1 Summary 47
7.2 Predicting Superlinear Speedup 48
7.3 Conclusions 50
7.4 Future Work 51

REFERENCES 53

APPENDIX A. Source Code 55

APPENDIX B. Detailed Results 64

APPENDIX C. System Descriptions 75

BIOGRAPHY OF THE AUTHOR 78

LIST OF TABLES

Table 3.1 CRAFT Benchmark Results 21

Table 6.1 Program 3 Results, Comm Interval = 10 Iterations 41

Table B.1 Program 1 Results, Two Processors Per Node 64

Table B.2 Program 2 Results 66

Table B.3 Program 3 Results 68

Table B.4 Program 1 Results, One Processor Per Node 70

Table B.5 Program 3 Results, Comm Interval = 25 Iterations 71

Table B.6 Program 3 Results, Comm Interval = 50 Iterations 72

Table B.7 Program 3 Results, Comm Interval = 75 Iterations 72

Table B.8 Program 3 Results, Comm Interval = 100 Iterations 73

Table B.9 Program 3 Results, Comm Interval = 250 Iterations 73

Table B.10 Program 3 Results, Comm Interval = 500 Iterations 74

LIST OF FIGURES

Figure 1.1	Moore's Law - Transistors Per Chip, [1]	2
Figure 1.2	Intel Processor Clock Speeds by Year, [2]	2
Figure 2.1	Speedup	9
Figure 2.2	Speedup for N=1024	10
Figure 3.1	CRAFT Time Components (not to scale)	21
Figure 5.1	Program 1 Results, Two Processors Per Node	35
Figure 5.2	Program 2 Results, Two Processors Per Node	35
Figure 5.3	Program 3 Results, Two Processors Per Node	36
Figure 5.4	Program 1 Results, One Processor Per Node	37
Figure 6.1	Program 3 Results, Comm Interval = 10 Iterations	41
Figure 6.2	Summary of Additional Program 3 Tests	42
Figure 6.3	Additional Program 3 Tests, Detailed View	43
Figure 6.4	Multiple Array Test Results	45

CHAPTER 1

Introduction

1.1 Growing Computational Power

Since the advent of the first microprocessor, the Intel 4004, microprocessor speeds have increased at an exponential rate [1, 2]. This behavior was predicted by Gordon Moore in 1965, when he observed the exponential growth in the number of transistors per integrated circuit[1, 3]. This observation became known as Moore's Law, which Intel expects to hold at least until the end of this decade[1].

The actual rate of Moore's Law was originally about a 12-month doubling time. This rate slowed down to about an 18-month doubling time in the 1970's, which has stayed relatively constant[4]. A graph of the number of transistors in various Intel microprocessors is seen in Figure 1.1. Note that in Figure 1.1 we have "connected the dots" in order to make the trend easier to see. As you can see in this graph, the number of processors in Intel chips has been increasing exponentially.

What this ability to pack more transistors onto a microprocessor means is that, among other things, the newer chips can have more registers, larger on-chip memory cache, wider data paths, and more logic circuits. It also means that memory and logic components can be placed closer together, allowing for a greater operating speed due to a shorter electrical path[4]. All of this leads to faster and faster processors, at relatively constant costs. This means that although the fastest processor available today costs about the same as the fastest processor available two years ago, it is more than twice as fast.

We have also seen clock speeds increase at a dramatic rate, and with more complexity on new microprocessors, as described above, they are able to do more with

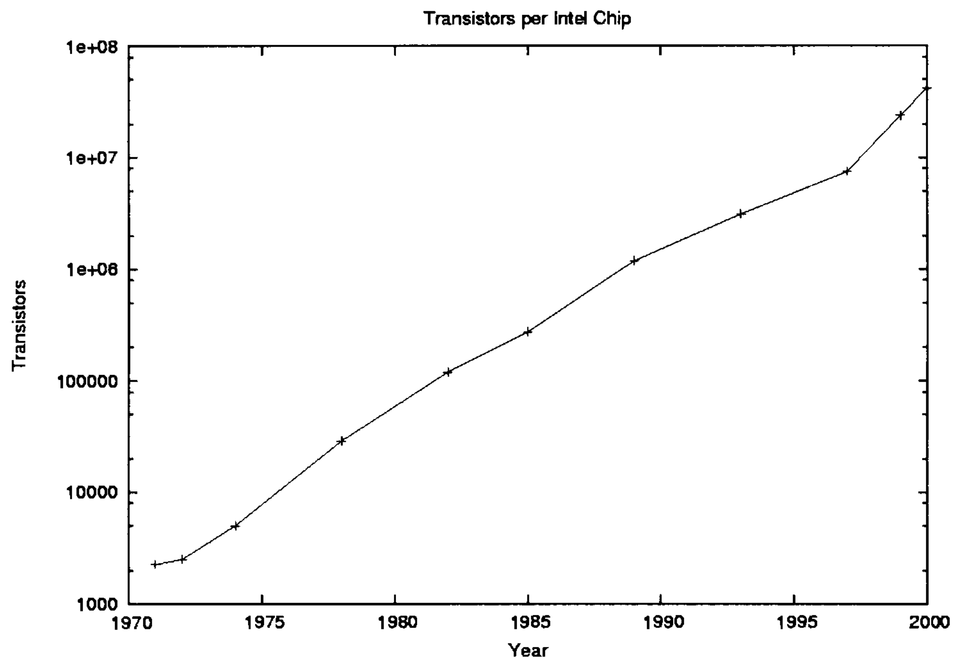

Figure 1.1: Moore's Law - Transistors Per Chip, [1]

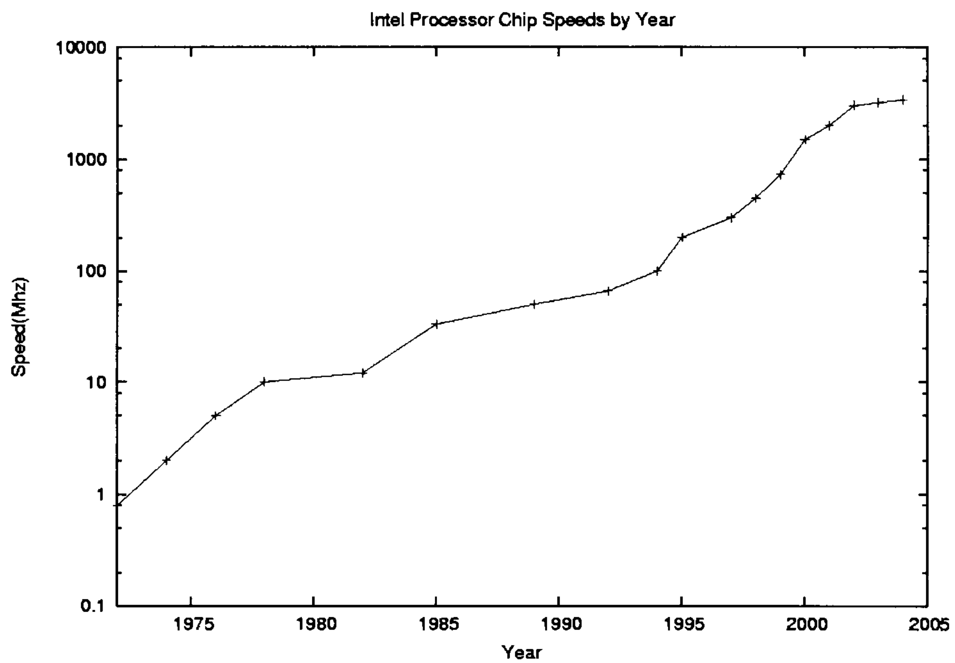

Figure 1.2: Intel Processor Clock Speeds by Year, [2]

each additional clock cycle. Figure 1.2 shows the clock speeds, at the date of introduction, of many popular Intel microprocessors. Like Figure 1.1, we have connected the dots in Figure 1.2. It can easily be seen from this graph that clock speeds have been increasing exponentially as well.

1.2 Growing Demand for Computational Power

Despite this relentless increase in computational power available in commodity microprocessors the world's thirst for CPU cycles remains unquenched. Because there is always the desire to squeeze more detail out of computer models, and because scientific data sets are getting larger and larger, scientist's computational needs have kept up with, or possibly even out-paced Moore's Law. For example, increased computational power allows for more detailed Earth climate models, and therefore more accurate weather prediction.

Even though we have this great demand for computational power, we are limited by current state of the art microprocessor manufacturing techniques. One can go to a local electronics store and purchase a microprocessor that executes over 3 billion operations per second. The price of this chip will be less than $500. However, one can not purchase a commodity chip running ten times faster at any price. Such a thing simply does not exist.

One can use specialized, and expensive, vector processors to achieve better speeds for large scientific calculations, but even single vector processors are not fast enough to satisfy the computational requirement of many of today's scientists and engineers. The only options available in this situation are to either wait for processor speeds to increase, at their current predictable rate, or utilize many processors together on the same problem. The latter approach is called parallel processing.

1.3 Parallel Computing Overview

In parallel processing the problem is essentially split up so it can be worked on in parallel by many different processors at the same time. This problem decomposition can be done either *a priori*, or during run time. Different problems will lend themselves to different decomposition methods.

For example, Computational Fluid Dynamics (CFD) codes lend themselves to the *a priori* decomposition. In these types of programs the global grid is decomposed into sub-grids for each processor[5]. When the program starts, each processor reads its own input files and performs the computation on its portion of the grid. Then each processor communicates with the processors computing the neighboring sub-grids to exchange boundary information after each iteration[5].

A common method of run time decomposition is through a replicated worker type algorithm. In this type of parallelism, there is a task-pool and a number of workers that retrieve tasks from the task-pool in parallel. When a worker finishes processing a task it may add a new task to the pool. The program will run until the task-pool is empty. This is a common method of parallelism for combinatorial problems like graph or tree searches[6].

Another method of parallelism that does not involve decomposing input problem, but instead involves distributing the functionality of the program. This type of parallelism is called pipelining, or pipelined computation[6]. In this approach to parallelism, data flows from one processor to another, and at each processor a different portion of the overall computation is to be done. Efficient pipelining requires keeping the pipeline full. If there is only one chunk of data to process then the pipeline will not provide any parallelism.

In parallel processing there are two major system architectures. The first is a shared memory parallel system. Historically these systems often used specialized vector processors, as was the case with the popular Cray supercomputers of the past[7]. This

shared memory model has been the traditional approach to supercomputing. The second major approach, that has steadily been gaining popularity for a decade, is a parallel system based on commodity microprocessors, usually with a distributed memory architecture. Distributed memory supercomputers based on commodity microprocessors and interconnects are commonly referred to as "Beowulf Clusters", in reference to the first cluster of this type, named Beowulf. The original Beowulf Cluster was built in 1994, when a research group at NASA had the need for a supercomputer, but could not afford a traditional one[7, 8].

With the rapid increase in the computing power of desktop and workstation computers, this approach has become a very powerful yet cost effective alternative to the traditional supercomputer[9]. In fact, for over a decade, the rate at which desktop and workstation processors have increased in performance has been greater than the rate at which traditional supercomputing processors, such as vector processors, have increased in performance[10].

Corresponding to these two major system architectures are two major programming paradigms for parallel processing. The first is the threaded model, where all the processors have access to the same shared memory. A common method to achieve shared memory parallelism is to use OpenMP directives. These directives give the compiler directions on how it can parallelize the code. The directives include telling the compiler which loops can be done in parallel, and which variables need to be local or shared in the parallel sections. Synchronization points and critical sections can also be specified. The compiler then creates threaded code based on these directives. The other major model is the distributed memory model, in which the programmer explicitly shares data between processors via message passing. The *de facto* standard for creating distributed memory parallel software is through Message Passing Interface (MPI) calls, where the programmer uses calls such as `MPI_SEND` and `MPI_RECV` to share data between processors[11].

1.4 Our Work

Today, at the University of Maine, researchers use a cluster supercomputer, similar in concept to the original Beowulf Custer but thousands of times faster, to model airflow over missile bodies, water flow in nanotubes, the carbon cycle in the Pacific Ocean, and to perform other computationally intensive calculations.

However, parallel processing is not trivial, and not all problems are well suited to this type of approach. Amdahl's Law, developed in 1967, is an equation showing that the inherently serial portions of a computation place an upper-limit on the potential speedup of the problem [6, 12, 13, 14]. Furthermore, Amdahl's Law places an absolute limit of N on the potential speedup, where N is the number of processors used in a parallel calculation. This upper-limit represents an ideal problem that is infinitely parallelizable.

Not only does this upper limit of N represent a infinitely parallelizable problem, it also does not account for any additional overhead that may be required to parallelize the computation[14]. With serial tasks and additional overhead achieving speedups even close to N would be impossible for many parallel calculations.

While conducting research for the SDMT (Supercluster Distributed Memory Technology) research project at the University of Maine, we saw speedups of a particular parallel computation that seemed to defy Amdahl's Law. This discovery caught our interest in the potential of speedups greater than N on N processors, which at first glance strikes one as very counter intuitive.

This work focuses on the implications of Amdahl's Law, and on some shortcomings of the law. It discusses situations where speedup greater than N is possible when a parallel computation is performed on N processors. This phenomenon is sometimes referred to as superlinear speedup.

CHAPTER 2

Significant Prior Research

2.1 Amdahl's Law

In 1967 Gene Amdahl, a researcher in IBM's mainframe division, wrote a paper, [12], promoting the uni-processor approach to computing. In this paper, Amdahl had observed that commonly 40 percent of executed instructions in typical programs of the time dealt with data management overhead. It was Amdahl's position that this could be reduced by a factor of two, and that it was highly unlikely that it could be reduced by a factor of three. Given that this overhead was sequential in nature, Amdahl stated that maximum speedup would be five to seven times the sequential rate.

This idea of serial overhead of 13.3% to 20% (assuming that the 40% overhead can be reduced by a factor of two to three) limiting maximum speedup to five to seven times was commonly generalized and reformulated to what is commonly known as Amdahl's Law, seen in Equation 2.1. Here S is the percentage of instructions sequential in nature, P is the percentage of parallelizable instructions, and N is the number of processors used in a parallel calculation[6]. This equation relating serial portions of code to speedup does not explicitly appear in Amdahl's work.

$$Speedup \leq \frac{S + P}{S + P/N} \tag{2.1}$$

If we look at 2.1 we can see that since P and S are percentages and add up to 1, then we can substitute $1 - S$ for P in 2.1:

$$\begin{aligned} Speedup &\leq \frac{S + 1 - S}{S + \frac{1-S}{N}} \\ &\leq \frac{1}{S + \frac{1-S}{N}}. \end{aligned} \tag{2.2}$$

If we are then to assume that the problem is infinitely parallelizable ($S = 0$), then we get the following upper-limit for speedup:

$$\begin{aligned} Speedup &\leq \frac{1}{S + \frac{1-S}{N}} \\ &\leq \frac{1}{0 + \frac{1-0}{N}} \\ &\leq \frac{1}{\frac{1}{N}} \\ &\leq N. \end{aligned} \tag{2.3}$$

This is very intuitive, and can be compared to many physical examples. For example, consider the task of digging a moat around a medieval castle. One hundred workers would complete the task in about one-hundredth of the time it would take a single worker, assuming they all work at the same rate, but 100 identical workers would never complete the task over 100 times faster than the single worker. This task of digging a moat would have a very small S value (almost zero), and speedup would be about N until the laborers are so numerous that they are getting in each other's way. Amdahl's Law would predict a maximum speedup of almost N, even for large values of N. Once the laborers are getting in each other's way we start seeing diminishing returns for each additional laborer added to the task. Given the small value of S and the correspondingly large value of P, a speedup of several thousand times would be possible.

Now consider the example of digging a well in the courtyard of this same castle. In comparison, this task would have a rather large S value. While multiple laborers may dig the well at one time, this number is quite small and depends on the diameter of the well. The small number of laborers that can fit in the well at one time corresponds to the small P value of the task. The depth of the well vs the diameter would correspond to the large S value of the task, as dirt cannot be removed until all the dirt above it has been removed.

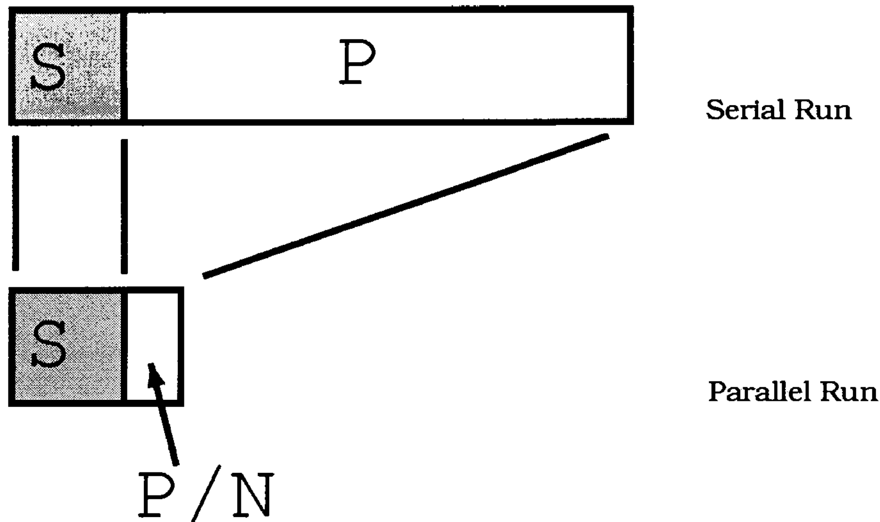

Figure 2.1: Speedup

In Figure 2.1 we see a graphical illustration of Amdahl's Law at work. One can see that adding more processors to the problem shrinks the parallel portion, and the runtime becomes dominated by the serial portion of the code. No matter how many processors are added the amount of time spent in the serial portion remains constant.

Figure 2.2 shows a graph of Equation 2.1, where $N = 1024$. One very important thing to note about this graph is the slope of the curve near $S = 0$. The slope of this curve is approximately $-N^2$, which tells us that only a limited number of problems would even experience a speedup of 100[13]. What seems like a reasonable serial percentage of 5% ($S = 0.05$) would limit our maximum speedup on 1024 processors to 20 times. In most cases it would make no sense to run such a problem on that number of processors, since the efficiency is so poor.

2.2 Gustafson's Scaled Speedup

In 1988 researchers at Sandia National Laboratories achieved what they felt were unprecedented speedups on a 1024-processor hypercube, and John L. Gustafson wrote a paper, [13], where he proposed something called *Scaled Speedup* to address their

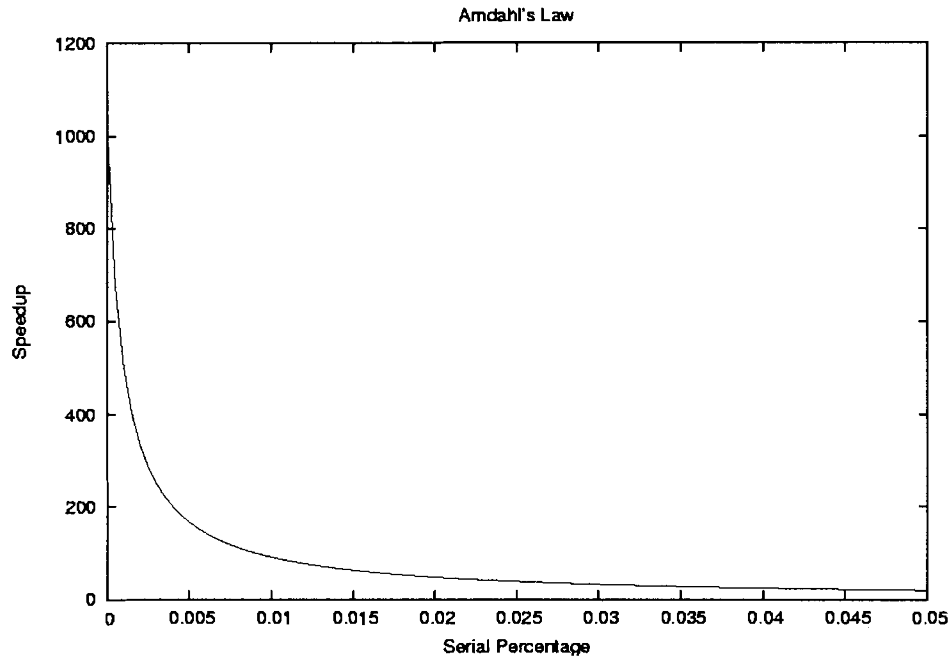

Figure 2.2: Speedup for N=1024

findings. Gustafson said that they saw speedups, using his scaled speedup model, of 1016 to 1021 on three problems with S values ranging from 0.004 to 0.008. From Equation 2.1 we can see that if $S = 0.004$ and $N = 1024$ we would get a maximum speedup of slightly over 201, which is much lower than the speedup that was observed by Gustafson.

Amdahl's Law assumes a fixed problem and variable run time. Instead, Gustafson argued that a more realistic scenario would be problems expanding to make use of an increased number of processors. In Gustafson's model, *Scaled Speedup*, it is the run time that is fixed, and the problem size is scaled when run on more powerful computers.

Gustafson had observed that it is usually the parallel part of the program that scales with problem size, but the S component grows much slower, if at all, as the problem size grows. By using this fact, Gustafson and his group created new, larger, problems that would run on the 1024-processor hypercube in the same wallclock time

that the original serial problem took. Gustafson had found that his three real world problems had parallel portions that scaled by 1023.9969, 1023.9965, and 1023.9965 when scaling the problems by 1024. This means that almost all of the additional work took place in the parallel portions of the code. This problem scaling is done by means of increasing the grid resolution, using a smaller time-step, adding more parameters, or other similar methods of extracting more detail out of the computer model.

Gustafson then used P' and S' to represent the parallel and serial time spent on the parallel system, respectively. Gustafson set $P' + S' = 1$ for algebraic simplicity, basically making P' and S' percentages of the run time of the parallel program, much like in our previous definitions of Amdahl's Law where P and S were percentages of the *serial* run time. The extrapolated run time on a serial system would then be $S' + P' * N$. Using this reasoning, the researchers derived their alternative to Amdahl's Law:

$$
\begin{aligned}
ScaledSpeedup &= \frac{S' + P' * N}{S' + P'} \\
&= S' + P' * N \\
&= N + (1 - N) * S'.
\end{aligned}
\tag{2.4}
$$

This equation, 2.4, became known as Gustafson's Law, and has been widely used to justify massively parallel processing[15]. However, it really is not a new law. If you recalculate the S value based on the new scaled parallel percentage, Amdahl's Law would predict similarly large speedups. This equivalence of Gustafson's Law to Amdahl's Law was proven mathematically in [15]. In this paper Yuan Shi gives a formula to translate the non-scaled serial percentages to the scaled serial percentages. Using the scaled serial percentage, Amdahl's Law gives speedups for Gustafson's three problems consistent with his scaled speedup calculations.

With Gustafson's fixed time, scaled speedup, approach we are able to see that even if massively parallel computing is not efficient for a given code and problem pair, massively parallel computing could be efficient for the same code given a larger problem. In his fixed time model the overall run time will be about the same, but the amount of work done in that time period can be thousands of times larger.

2.2.1 Scaled Speedup Explained

In most cases the amount of computation required for the "main loop" of a program is proportional to a power of the size of the input data[6, 13]. The initialization of the program is generally proportional to the size of the input data[6]. Since the initialization portion of a program often contains the majority of a program's sequential instructions, and since the complexity of the computational portion of code grows faster than the complexity of the initialization, in most cases one can decrease the serial percentage in Amdahl's law by using a larger input data set (as long as the computation is highly parallelizable)[6]. This makes the scaled speedup approach a very useful technique.

Equation 2.5 shows how the serial percentage in Amdahl's Law can be described as a function of the amount of computation required for initialization and the amount of computation required for the "main loop"[6]. It is assumed that the initialization code is sequential in nature, and that the remainder of the program can be highly parallelized. In this example the amount of computation required for the initialization is cn, and the amount of computation required for the main loop is dn^2, where c and d are constants and n is the size of the problem. Note that this is just an example, and dn^2 was arbitrarily chosen. We could have easily chosen dn^3, or any other power. From Equation 2.5 we can see that increasing the input problem will reduce the serial percentage of this program.

$$S = \frac{cn}{dn^2} = \frac{c}{dn} \tag{2.5}$$

2.2.2 Fixed Time vs Fixed Size

The fixed time scaled speedup approach by Gustafson validated the approach of massively parallel processing, however, this fixed time approach does not work for all problems. Consider the example of a weather prediction model. Assume a serial implementation of a climate model takes thirty days to forecast the weather three days later. By the time the model is done running the data is useless. In this case the efficiency of the problem is secondary to the time it takes to run the problem. For example, even if the model achieved a speedup of 30 by running on 64 processors it would make sense to run the model on this number of processors in order to have the weather forecast in time for it to have value.

The example of a weather model can also be used as an example where the fixed time approach makes sense. Say with a current weather model, the three day forecast can be computed in less than one day. With increased computational power it might make more sense to add more factors into the weather model or increase the grid resolution to get a more accurate solution instead of performing the calculation in less time[16].

If we desire more computational power in order to achieve better accuracy, the fixed time, scaled speedup approach makes sense[13, 16]. If we desire more computational power in order to arrive at a solution faster, the fixed problem size approach makes the most sense. Unfortunately the fixed problem size approach rules out massively parallel computation for many problems, and this way of thinking led most early supercomputers to be built with a small number of processors[16].

2.3 Parallel Overhead

It has been observed that Amdahl's Law could even be overly optimistic due to overhead incurred while parallelizing the code[14, 16]. This overhead is often called *parallel overhead*, and it includes additional code required to parallelize the task, often in the form of MPI calls in the distributed memory model or thread control code in the

shared memory architectures, as well as communication latency in distributed memory supercomputers and ensuring cache coherence in cache coherent (cc) NUMA (Non-Uniform Memory Access) architectures[6, 17]. Non-uniform memory access means that not all memory can be accessed in the same amount of time. NUMA machines may locally cache data located in "remote" memory (memory not local to the processor, meaning it takes more clock cycles to access). Cache coherence means that when data is changed, all cached copies of that data have to be changed to reflect the new value thereby maintaining "cache coherence"[6, 17].

Robert G. Brown proposed a new estimate of speedup based on Amdahl's Law that accounted for some of the parallel overhead[14]. In his new estimate, Brown took into account additional time doing serial tasks (such as interprocessor communication) and additional time spent by each processor doing additional parallel tasks (such as additional setup tasks required on each processor for the parallel version of the code). A new equation for speedup can be seen in 2.6. It is based on the formula that appeared in [14] but with variable names changed to reflect the notation established in Equation 2.1 and with $S + P$ normalized to one rather than reflecting the actual total runtime. Here we use $\hat{S}$ to signify additional time doing serial tasks, and $\hat{P}$ to signify additional time doing parallel tasks.

$$Speedup = \frac{S + P}{S + N * \hat{S} + P/N + \hat{P}} \tag{2.6}$$

From Equation 2.6 we see that actual speedup will likely be less than that predicted by Amdahl's Law.

2.4 Superlinear Speedup

Superlinear speedup is the term commonly used to refer to speedup greater than N when a parallel calculation is performed on N processors. According to Amdahl's

Law this is an impossibility. Also, according to [15], since every practical parallel program must consolidate the final answer the serial percentage is never zero, making even a speedup of N when running on N processors is impossible. Historically claims of superlinear speedup have often been due to inefficient serial algorithms[18, 19].

One of the earliest "proofs" of superlinear speedup was a 1986 short paper appearing in the journal *Parallel Computing* by D. Parkinson,[20], where he asked us to consider the following code fragment:

```
DO I = 1,N
A(I) = B(I) + C(I)
CONTINUE
```

Parkinson argued that by running this code fragment on N processors the loop overhead could be eliminated, thereby causing a speedup of greater than N.

It is interesting to note that his paper, called "Parallel efficiency can be greater than unity", was accompanied by a paper called "Superlinear speedup of an efficient sequential algorithm is not possible" in the same July 1986 issue of *Parallel Computing*. These two conflicting articles were even appeared back to back in the journal!

In [21] a model of parallel computation capable of explaining speedups greater than N on N shared memory processors is explained. The reasons given for speedup greater than N include: the sequential algorithm is somehow constrained to use an inferior method; the problem is NP-hard and the best known algorithm is a randomized search (when multiple choices are explored in parallel the probability that they all lead to lengthy calculations is low); the parallel calculation may have reduced overhead; the multiprocessor system has an increased cache size; and the parallel calculation hides latency.

Of these causes of superlinear speedup the first two are self explanatory, and not particularly interesting. In fact, Helmbold *et al* provide references to claims that while

the second cause (the randomized search case) is possible, it has not been observed in practical algorithms, and their model does not account for this cause of superlinear speedup.

The latency hiding technique can be used on the uni-processor model, and can be considered an optimization and not a source of superlinear speedup[21]. The speedup greater than N caused by reduced overhead applies to shared memory machines, as they state that the cost (i.e., processor time) of some system calls on an n processor machine will be $1/nth$ the cost on a uni-processor machine. The idea that more processors allow the system overhead to be spread out does not apply to distributed memory cluster computers, since each node in a cluster computer has its own operating system.

The idea of speedup greater than N due to an increased cache is quite interesting. However, like the reduced overhead cause, the arguments they present are based on a shared memory model[21]. They state that the cache miss ratio may decrease as n increases because the number of different tasks that each processor must execute will be reduced[21]. On a distributed memory cluster, such as the platform we use, this is not true. In particular, we know that each node used in a parallel job will be running the same number of processes per processor regardless of the number of nodes used in the computation.

In [18], Gustafson says that in some cases performance can increase instead of decrease as the problem size per processor shrinks. Remember that in [13] Gustafson argued for scaling the global problem size up as the problem is run on more processors in order to achieve better efficiency. In [18] he is looking at the case where the global problem is fixed, and therefore the local problem shrinks as it is run on more processors. Gustafson points out that the different speeds of tiered memory in distributed memory supercomputers could allow for superlinear speedup (not caused by an inefficient serial algorithm). Gustafson offers no real world problem that demonstrates this sort of speedup.

In this same paper, Gustafson offers another cause of superlinear speedup, which he calls "Changing Routine Profile"[18]. In this cause of superlinear speedup, running on more processors allows more time to be spent in faster routines. Gustafson assumes that the "fixed time" approach is being used, and he gives experimental results showing a speedup of 4.16 when using four processors and fixing the run time of the example program at one minute.

We don't find this cause of superlinear speedup particularly interesting because basically it is simply an effect of imposing a time limit on the computation and is not related to the supercomputer architecture. This can be explained using Gustafson's own example of this phenomenon.

Gustafson gives a physical example of moving a piano with a time limit of 30 minutes, and work measured in distance moved[18]. In this example he states that with a single mover, the piano might be moved a few feet out the door, while a truck idles outside. With two movers, the piano might be moved outside, loaded onto the truck, and driven 20 miles down the highway. By adding a second mover, the amount of work done was increased by several thousand times (a few feet compared to 20 miles). Using two movers to move the piano out of the house and onto the truck might be 1.9 times faster than a single mover, and once they are in the truck the speedup of having a second mover is one, meaning it is the same speed (the second mover obviously cannot make the truck drive any faster). By imposing the fixed time, that speedup of 1.9 makes a huge impact on the amount of work done, but if we instead measured how long it takes to move a piano from one location to another with one and two movers, we would see that two movers is not more than twice as fast because for part of the task they are 1.9 times faster, and for the remainder of the task they are the same speed. Therefore, imposing an artificial fixed time limit that is not sufficient to solve the problem can lead to artificial examples of superlinear speedup.

In the past fourteen years the Helmbold[21] and Gustafson[18] papers have received limited attention. In searching for citations of these papers, only a handful were found. Most often citations of these papers were used to justify minor superlinear speedups, with little or no explanation of the actual mechanism responsible for the particular instance of superlinear speedup. Furthermore most claims of superlinear speedup occurred in shared memory supercomputers, and no proof of large scale superlinear speedup in Beowulf-style commodity based clusters was found.

A typical reference to either of these papers is exemplified by a paper on a parallel radiosity algorithm, [22], which uses Gustafson's work[18] to justify their speedups that were slightly larger than 16 on 16 processors. These results were obtained on a Silicon Graphics Origin2000, a shared memory supercomputer. We have yet to find reports of significant superlinear speedup on large scale distributed memory supercomputers.

2.5 Chapter Conclusions

In this chapter we have summarized many of the views on parallel processing speedup. Amdahl's pessimistic view on parallel computation is one of the oldest, most intuitive, and most well-known of these views. It has also been said that Amdahl's law might even be too optimistic because of "Parallel Overhead". These beliefs ruled out massively parallel processor for many problems, but Gustafson saw that if, instead of fixing the problem size, we fixed the run time many more problems could see very efficient speedups on large processor counts. By fixing the run time, faster computers allowed larger or more complex problems to be used, which effectively shrinks the serial percentage of total run time spent in serial code. This was used to justify massively parallel processing, and soon systems with hundreds or even thousands of processors became much more common. The goal of these larger systems was often not to complete a problem faster, but instead to run larger, more detailed, or more accurate simulations.

While scaling the problem size allowed many programs to be run utilizing more processors, the absolute upper limit of speedup was still considered to be N. This limit was based on the assumption that the computer performed linearly as problem sizes were reduced. Prior research has shown that small superlinear speedups could occur on shared memory supercomputers for a variety of reasons. Prior research has also suggested that superlinear speedup would be possible in a distributed memory supercomputer. In the following chapters we will show examples of superlinear speedup in a distributed memory cluster computer, and we will identify the properties of both the processors and programs that allow this to happen.

CHAPTER 3

Research Questions

3.1 Problem Introduction

As stated in Chapter 2 we know that the number of serial tasks in an algorithm severely limit its potential speedup when performing the computation in parallel. We also know that performing a computation in parallel often introduces additional overhead, as described in [14]. Because of this, while working on the University of Maine Supercluster Distributed Memory Technology (SDMT) research project, we were initially surprised to see speedups of 4.2 and 8.53, when increasing the number of processors used on a particular parallel computation by four and eight times respectively. Since Amdahl's Law is a law of diminishing return, one would expect doubling the number of processors would result in a less than doubling effect on the speedup, and that each additional doubling of processors would see a less efficient speedup than the previous doubling.

The SDMT research group deals primarily with a package called CRAFT CFD, available from (and a registered trademark of) CRAFT Tech of Pipersville, PA. This code, henceforth simply referred to as CRAFT, is a state-of-the-art, three dimensional structured grid Navier-Stokes code. More information on CRAFT Tech and the CRAFT code can be obtained from the CRAFT Tech website, http://www.craft-tech.com.

When performing CRAFT benchmarks, we usually compare something called an "iteration time", which is the time it takes for one pass through the main CRAFT loop. The iteration time consists of a computational portion, and a communication portion. For timing purposes we break a CRAFT run into several portions: startup time, many iterations, a write time, and end time (the time between the end of the write portion and actual program termination). An actual CRAFT run time is dominated by the iteration

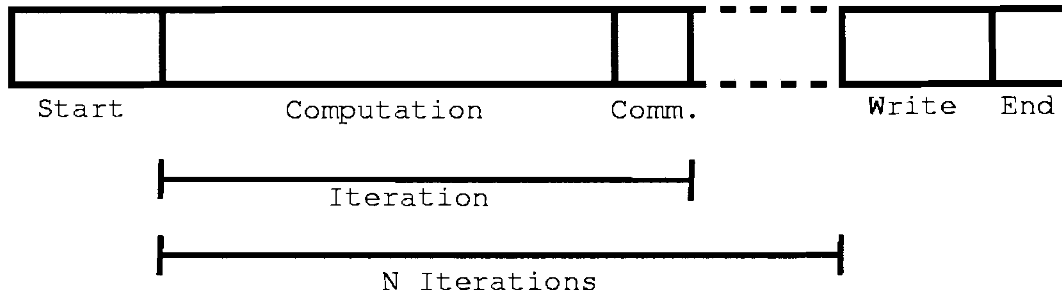

Figure 3.1: CRAFT Time Components (not to scale)

Table 3.1: CRAFT Benchmark Results

Number of CPUs	Iteration Time (s)	Theoretical Maximum Speedup (from 32 CPUs)	Measured Speedup
32	138.2	1	1
64	69.7	2	1.98
128	32.9	4	4.2
256	16.2	8	8.53

times, so therefore this is the number we are most interested in. Figure 3.1 shows the various components that are timed during a CRAFT run. Note that the lengths of each portion do not accurately reflect their overall percentage of a complete run time (for example, communication time is a tiny fraction of an iteration time).

In Table 3.1 we see iteration times for a CRAFT benchmark problem run on 32, 64, 128, and 256 CPUs. We see that when doubling the number of processors from 32 to 64 the resulting speedup is 1.98, slightly less than the theoretical maximum speedup of two when doubling the number of processors. This speedup of nearly two when doubling the number of processors is not surprising since the iteration is very parallelizable. However, in the 128 and 256 CPU cases, we see speedups of 4.2 and 8.53 when compared to the 32 CPU case. Both of these speedup values are larger than the theoretical maximum speedup. Even though these are the iteration times, and not timing data for the entire program, Amdahl's Law should apply to subsets of the code, otherwise it could be broken.

After thinking about these results, we soon identified an assumption made by Amdahl's Law that we felt would allow for speedups larger than N if the assumption were not true.

3.2 Our Hypothesis

3.2.1 Amdahl's Assumption

Amdahl's Law implicitly assumes that the processing time scales linearly with problem size. However, the program's input data is changing as the problem is decomposed further to take advantage of more processors. This is especially the case where each processor works on a smaller sub-problem, such is the case with CFD codes.

3.2.2 Problem with this Assumption

We know that a processor does not always operate at the same speed with different input data. Different input may cause the program to take a different path through its instructions, varying the run time. Also the change in volume of data can make a significant impact in the speed of a program. Certainly a program that has to process half as much data should complete in less time, but in most cases it will not reduce the crunch time by more than half, where we define crunch time as the time spent processing the input data, but not doing other program "housekeeping" tasks. However, there are certain instances where halving the input data can cause a program's crunch time to be reduced to *less than* half of the original time.

This behavior has to do with tiered memory found in modern microprocessors. Typically a computer has a large amount of disk storage, a significantly smaller amount of Random Access Memory (RAM), and one or more levels of cache memory, which are much smaller than main memory[4]. By decreasing the input data so it fits entirely into RAM instead of having to work from disk we see a significant performance gain because

paging data to and from disk is much slower than the access time of RAM. The same holds true for RAM vs. cache memory. Typically supercomputers do not utilize virtual memory because of the performance penalty of swapping to and from disk. Problems must be run on enough processors so the local problem can fit entirely in physical RAM. Such is the case with the University of Maine supercomputers.

The supercomputer at the University of Maine, Blackbear, on which the CRAFT benchmarks in Table 3.1 were performed, is comprised of 1 GHz Pentium III processors with 256 kilobytes of L2 cache. In these PIII processors it takes about 7 clock cycles to fetch a 32-bit word stored in L2 cache, however, it takes about 60 clock cycles to fetch a word from RAM[23]. This makes the L2 cache almost ten times faster than RAM. Even though the amount of L2 cache is much smaller than data sets for typical real world supercomputer applications, the processor attempts to make the best use of the cache by not only pulling in one 32-bit word at a time, but pulling eight 32-bit words into cache at once (pulling a total of 256 bits, or 32 bytes of data)[23]. This group of data is called a *cache line*[24].

The rationale is that memory accesses are likely to be sequential, so once a line is pulled into cache the next several memory accesses will hopefully be of data already in cache[24, 25]. When pulling in 32-byte lines into cache the 256 kilobytes fill up quickly, so eventually older data will be pushed out of cache. Ideally one hopes to read in a cache line, and then get several "cache hits" in a row before the next "cache miss" when another line is read in. In this situation most memory accesses are made to the L2 cache with the access time roughly ten times faster than RAM.

3.3 Chapter Conclusions

We think that we could be seeing effects of our test platforms cache memory that caused CRAFT to run slightly faster (get a larger percentage of cache hits) on the smaller local problem. We also believe that the cache memory could allow for a

program to seemingly break Amdahl's Law by achieving *superlinear speedup.* Upon further investigation we discovered an explanation of superlinear speedups in shared memory supercomputers[21], which didn't apply to our cluster, and we found a claim that superlinear speedup due to tiered memory in distributed memory supercomputers was possible[18], but no proof was given.

In the following chapters, the existence of such programs (programs that show superlinear speedup due to cache memory) will be proven, thereby proving the conjecture made by Gustafson in [18], and key attributes will be identified with which one can predict the possibility of superlinear speedup in real world programs.

CHAPTER 4

Research Methods

4.1 Experiment Overview

Our experiments consisted of running several test programs on the Kearney supercomputer. Each test was run on a number of different processor counts ranging from one up to 124 (the maximum number of CPUs available for computations on Kearney at the time of this research). For each test program three runs were performed at each processor count and the average of those run times along with the actual three run times was recorded.

Timing was done with `MPI_Wtime` function, which returns a double precision floating point number. The value of this number is defined as the number of seconds that have passed since an arbitrary time in the past[26]. This value will be recorded at the start of the program, and again after the "main loop" is complete just prior to program termination. This timing will be done by MPI process 0, which will also be responsible for collecting and merging data from all the other processes. This data aggregation overhead will be included in the timing information.

4.2 Test Programs

Our base test suite consist of three similar programs. The programs all perform floating point operations over a large set of data that they make repeated passes through. The programs were designed to be trivially parallelizable, and can therefore easily be run on varying numbers of processors. The datasets that are used are generated at run time, and although the programs are not performing an interesting calculation, the result can be used to make sure the program produces the same result regardless of the number

of processors the calculation is performed on. The calculation over the dataset produces a single number than can be compared between all runs.

The complete source code for all the test programs can be found in Appendix A. Below you will find descriptions and pseudo code for each of the three programs.

4.2.1 Program 1

Program 1 is the most basic of our test programs. This program simply repeats a numeric calculation over and over on a large data set. The idea is that as the calculation and data set are split over more and more processors eventually the entire local dataset will fit entirely into cache. This program accesses the data sequentially, so it already sees much of the benefits of cache (typically a cache miss will be followed by several cache hits). As with all the the test programs the size of the data set, and the number of iterations in the “main loop” makes the initial “setup” code a very small percentage of the overall executed code. Communication was minimized in Program 1.

Program 1 Pseudo Code

```
Initialize MPI

Start Timing

DataSize = GlobalDataSize / NumProcs

Allocate myData[DataSize]

For i = 0 to DataSize-1 [
  myData[i] = 1.5
]

For i = 0 to NumIterations [
  sum = sum + myData[i] * constant
]

Partial Solutions Gathered by Node 0

End Timing
```

```
Print Results
```

4.2.2 Program 2

Program 2 is very similar to Program 1, except for the order the data is accessed. In Program 2, memory is no longer accessed sequentially, and the benefits of the cache should be minimized until the local data set completely fits into cache. This should amplify the "cache effect" and demonstrate remarkable speedup.

Program 2 Pseudo Code

```
Initialize MPI

Start Timing

DataSize = GlobalDataSize / NumProcs

Allocate myData[DataSize]

For i = 0 to DataSize-1 [
  myData[i] = 1.5
]

For k = 0 to NumIterations [
  For i = 0 to 4 [

    j = i

    while j < DataSize [
      sum = sum + myData[j] * constant
      j = j + 5
    ]

  ]
]

Partial Solutions Gathered by Node 0

End Timing

Print Results
```

4.2.3 Program 3

Program 3 introduces more communication into the "main loop", but otherwise is very similar to Program 2. In Programs 1 and 2, the local solutions are aggregated once at the end of the program, in Program 3 partial local solutions will be sent to mpi_node 0 at a preset interval. Initially this communication interval was set to every 1,000 iterations.

Program 3 Pseudo Code

```
Initialize MPI

Start Timing

DataSize = GlobalDataSize / NumProcs

Allocate myData[DataSize]

For i = 0 to DataSize-1 [
  myData[i] = 1.5
]

For k = 0 to NumIterations [
  For i = 0 to 4 [

    j = i

    while j < DataSize [
      sum = sum + myData[j] * constant
      j = j + 5
    ]

  ]

  If k mod CommInterval == 0 OR k == last iteration [
    Partial Solutions Gathered by Node 0
  ]

]

End Timing

Print Results
```

4.2.4 Data Set and Number of Iterations

The size of the global dataset was set at 1,500,000 double precision floating floating point numbers. A double precision floating point number is represented by eight bytes, giving our dataset a size of 12,000,000 bytes, or twelve megabytes (MB). This dataset is extremely small by modern supercomputing standards, but it allowed the local datasets to fit completely in cache at 47 processors. At that point the local dataset was just over 255,300 bytes, or about 255 kilobytes (KB). This allowed us a good number of runs with the local dataset larger than cache, as well as a large number of runs where the local dataset is smaller than the available cache. With a larger dataset we would simply need to utilize more processors to reduce the local dataset to a size that would completely fit into cache.

After the size of the data was set at 12 MB, we began to experiment with a varying number of iterations, or the number of passes through the dataset. After trying various numbers of iterations for Program 1 (starting at 100 for initial testing, and increasing by multiples of ten), we found that 1,000,000 iterations would take around thirteen hours to complete on one processor. The final number of iterations was fixed at 1,750,000.

For Program 2, which also used a data set of 1,500,000 double precision floating point numbers, we found 1,000,000 iterations took approximately sixteen hours. The reason this is longer than 1,000,000 iterations in Program 1 is because Program 2 sees little benefit from cache, while Program 1 accesses the data sequentially and sees a large cache hit percentage. The cache hit percentage in Program 1 on one processor is essentially 75% because with each cache miss a 32-byte line is pulled into cache (as discussed in Chapter 3). Since double precision numbers are eight bytes, a cache line can hold four double precision floating point numbers, meaning the next three will will pulled into cache along with the double that is currently being accessed. Therefore in Program 1 a cache miss can be followed by three cache hits, giving us approximately

75% hit rate (neglecting cache misses due to cache being flushed by context switches). Based on this, we decided to set the number of iterations for program 2 to 1,500,000.

Since Program 3 is essentially the same as Program 2 with the exception of the amount of communication, we used the running time for one processor from Program 2 as the running time on one processor of Program 3 since the communication in Program 3 is unnecessary on a single processor run. Program 3, like Program 2, performed 1,500,000 iterations over 1,500,000 double precision floating point numbers for the dataset.

4.3 Test Procedure

All three test programs were run a total of three times for each processor count and the average time was used for calculating the speedup. We performed a set of tests utilizing two processors on each node, allowing us to fully utilize the computing resources available on the Kearney cluster, and we also performed a set of test with Program 1 utilizing one processor per node while leaving the second processor idle. Comparing tests run using one processor per node and two processors per node allowed us to look for signs of memory contention affecting program speedup (the two processors in each Kearney compute node share the same memory bus).

4.4 Additional Minor Experiments

In addition to the three main test programs mentioned above, additional tests were run on a more limited number number of processor counts. The limited number of runs and processor counts of these additional minor tests was due to both the time required to make runs at hundreds of different processor counts, and as a courtesy to other cluster users.

4.4.1 Communication Interval Tests

Additional tests included additional limited runs of Program 3 (on a handful of different processor counts) with various communication intervals. Since these additional tests were used to clarify our understanding of results from the three major test programs outlined above, the results from these additional tests will be presented in the discussion of the results of the related major test.

4.4.2 Multiple Array Tests

Another set of tests were done to show that although our major test programs only used one array, superlinear speedup can be seen with programs that have multiple arrays. In these tests we created two test programs, one of which steps through two arrays at the same time, the other of which steps through one array, and then the other array. These tests were conducted on 1, 8, 16, 32, 48, 64, 80, 96, and 112 processors. Because of other jobs on the cluster, and because of time constraints, only one run was performed at each of these processor counts rather than averaging three runs at each processor count like we did for previous experiments. The programs used for these tests, called MultiArrayTest 1 and 2, were based on Program 2, and the source code appears in Appendix A. Like Program 2, these programs perform 1,500,000 iterations over their data, however, instead of one 1,500,000 element array they have two 750,000 element arrays. Since these tests only produce a few data points, the results will be presented along with their discussion in Chapter 6.

4.4.2.1 MultiArrayTest 1 Pseudo Code

```
Initialize MPI

Start Timing

DataSize = GlobalDataSize / NumProcs
```

```
Allocate myData[DataSize]
Allocate myData2[DataSize]

For i = 0 to DataSize-1 [
  myData[i] = 1.5
  myData2[i] = 2.5
]

For k = 0 to NumIterations [
  For i = 0 to 4 [

    j = i

    while j < DataSize [
      sum = sum + myData[j] * constant + myData2[i] * constant
      j = j + 5
    ]

  ]
]

Partial Solutions Gathered by Node 0

End Timing

Print Results
```

4.4.2.2 MultiArrayTest 2 Pseudo Code

```
Initialize MPI

Start Timing

DataSize = GlobalDataSize / NumProcs

Allocate myData[DataSize]
Allocate myData2[DataSize]

For i = 0 to DataSize-1 [
  myData[i] = 1.5
  myData2[i] = 2.5
]

For k = 0 to NumIterations [
  For i = 0 to 4 [

    j = i

    while j < DataSize [
      sum = sum + myData[j] * constant
```

```
      j = j + 5
    ]

  ]
]

For k = 0 to NumIterations [
  For i = 0 to 4 [

    j = i

    while j < DataSize [
      sum = sum + myData2[i] * constant
      j = j + 5
    ]

  ]
]

Partial Solutions Gathered by Node 0

End Timing

Print Results
```

4.5 Chapter Summary

In this chapter we described three major test programs, plus several additional minor tests, which we used to prove the existence of superlinear speedup due to microprocessor architecture and to better understand the underlying mechanisms that cause superlinear speedup in these cases. In the following chapter we present the results from the above mentioned experiments.

CHAPTER 5

Research Results

5.1 Introduction

This chapter presents the results from the tests described in Chapter 4. In-depth discussion of these results is reserved for Chapter 6. First we present the results from the three major test programs when utilizing two processors per node. These results are followed by our Program 1 tests utilizing one processor per node.

5.2 Results, Two Processors Per Node

First we did our major runs using two processors per node, allowing us to utilize all processors available for computation on the Kearney cluster. As discussed in Chapter 4, Section 4.3, we performed runs utilizing two processors per node for all test programs, and additionally we performed Program 1 tests utilizing one processor per node (leaving one processor idle) in order to look for signs of memory contention.

5.2.1 Program 1 Results

Figure 5.1 shows our results for our Program 1 tests. The detailed timing information from these tests can be seen in Table B.1, located in Appendix B.

5.2.2 Program 2 Results

Figure 5.2 shows our results for our Program 2 tests. The detailed timing information from these tests can be seen in Table B.2, located in Appendix B.

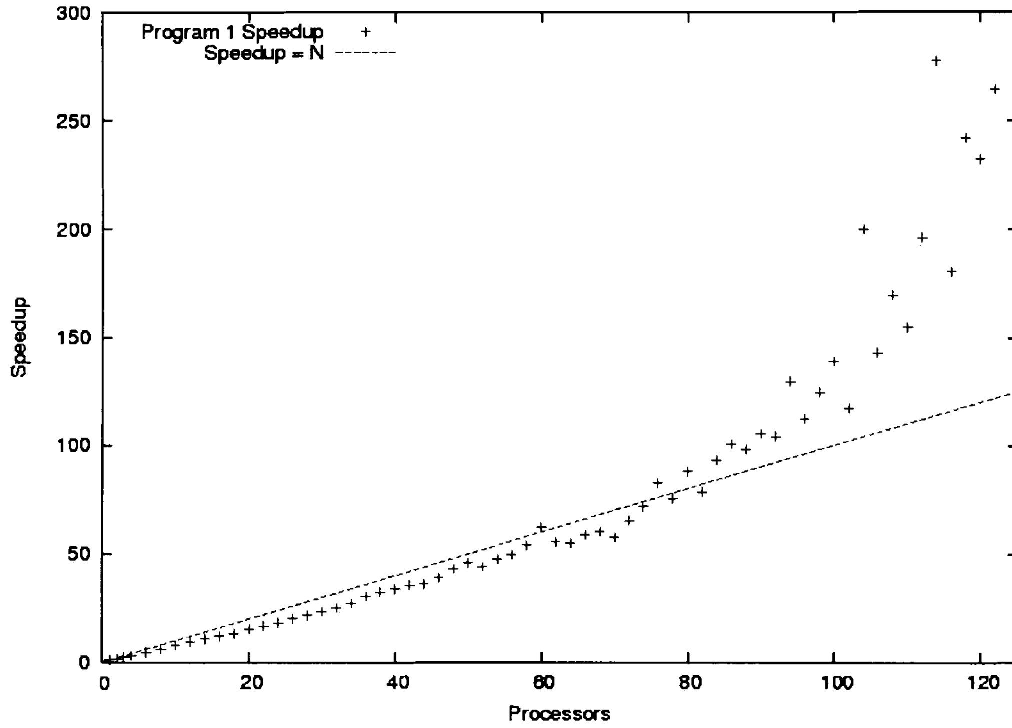

Figure 5.1: Program 1 Results, Two Processors Per Node

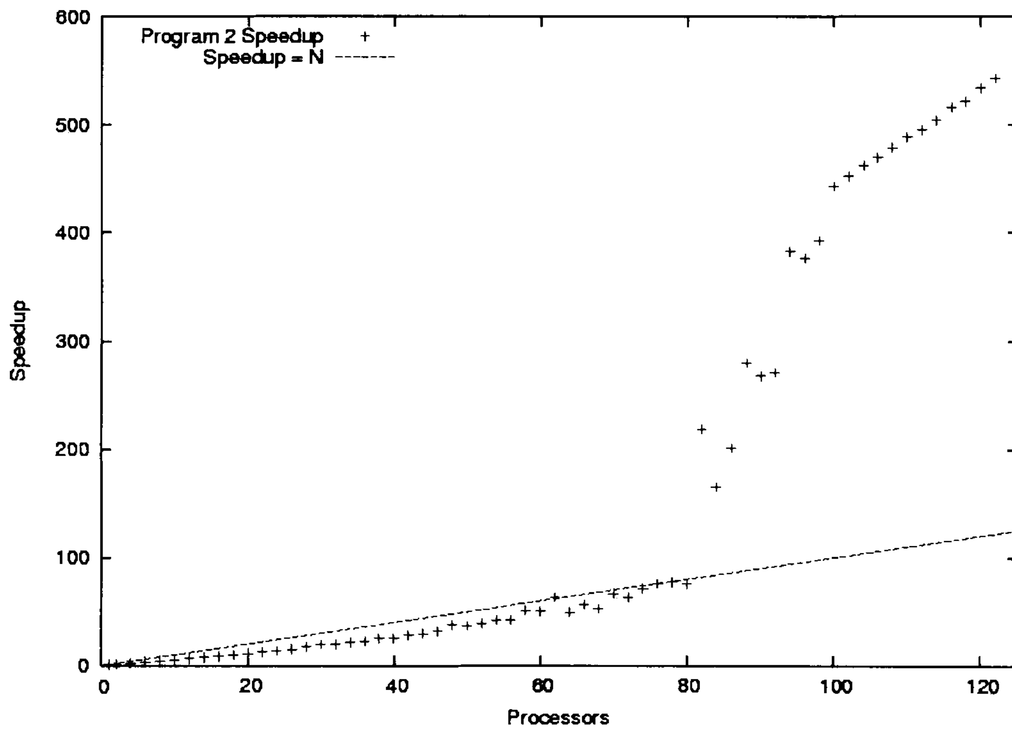

Figure 5.2: Program 2 Results, Two Processors Per Node

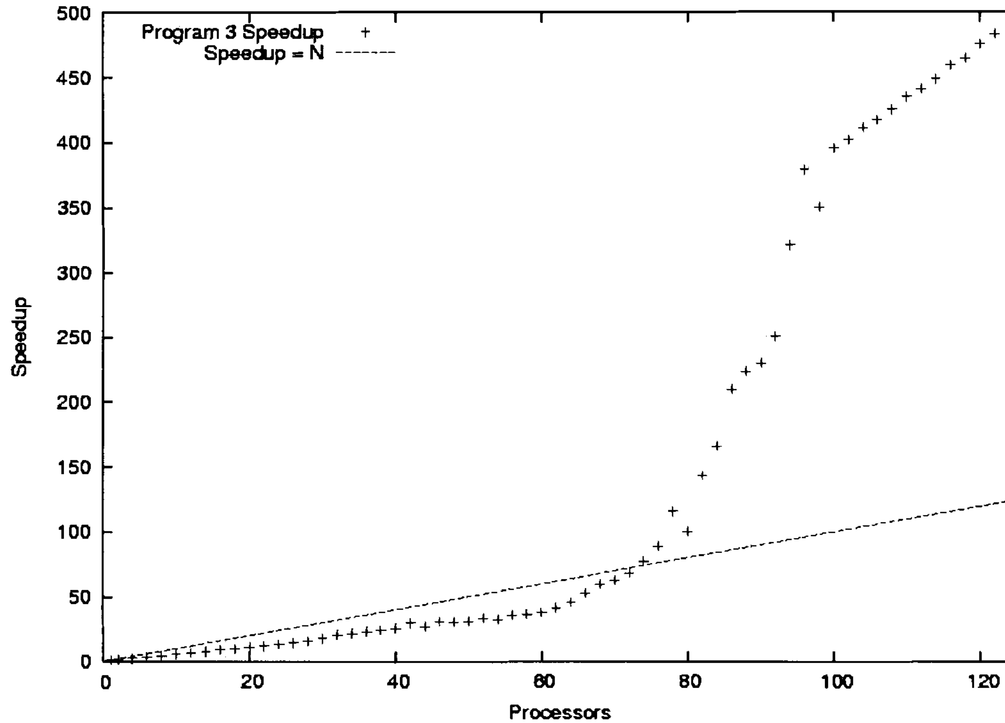

Figure 5.3: Program 3 Results, Two Processors Per Node

5.2.3 Program 3 Results

Figure 5.3 shows our results for our Program 3 tests. For these test we used an initial communication interval of 1,000 iterations. That is, after every 1,000 iterations through the data set, all nodes perform a synchronous communication to consolidate a partial result onto the node with MPI rank of 0. MPI rank is a unique number from 0 to N-1, where N is the number of processors used in the MPI job, assigned to each process in the job. The detailed timing information from these tests can be seen in Table B.3, located in Appendix B.

5.3 Results, One Processor Per Node

Our single processor tests of Program 1 consisted of reserving both processors available on each node taking part in a particular run, and only running one process on

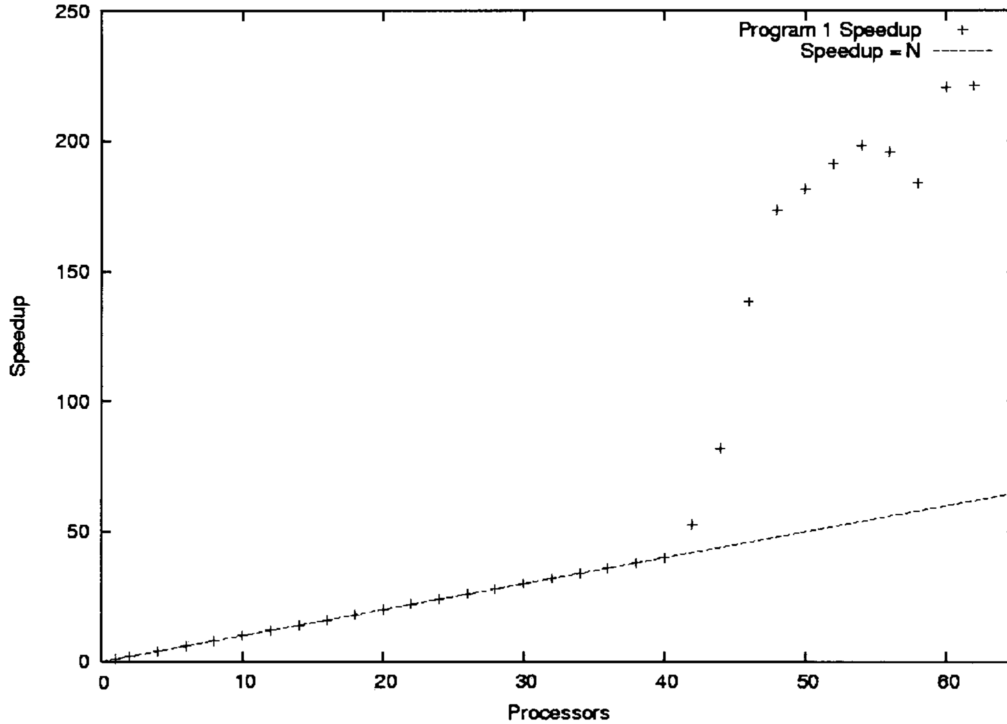

Figure 5.4: Program 1 Results, One Processor Per Node

that node. This allowed us to virtually eliminate memory bus contention, and contention for processor time by system processes. Table B.4, located in Appendix B, summarizes our results, and a the speedup has been graphed in Figure 5.4.

5.4 Chapter Conclusions

As can easily be seen in Figures 5.1, 5.2, 5.3, and 5.4 all of our major tests showed superlinear speedup. While we had expected to see superlinear speedup in all of these test programs, some aspects of the results did surprise us. In the following chapter we discuss these results in detail, and we also discuss the results from our additional minor tests, which were described in Chapter 4, that we conducted to better understand these results.

CHAPTER 6

Discussion

6.1 Program 1 Discussion

6.1.1 Two Processors Per Node

As seen from Figure 5.1 and Table B.1, Program 1 clearly achieved superlinear speedup. We can see that at 48 processors the local dataset fit entirely into cache on each processor, with a maximum local dataset size of 250,000 bytes. Recall that in the result tables the table listing the size of the local dataset lists the maximum size of any local dataset. If the global dataset does not divide evenly, then one processor will end up with a local dataset that is slightly larger than the rest of the processors (up to $(N - 1) * 8$ bytes larger, where N is the number of processors being used).

In Figure 5.1 and Table B.1, we can see that between 40 and 62 processors the measured speedup begins to grow faster as the number of processors (referred to as N for the remainder of the chapter) increases, and at 60 processors the measured speedup is greater than N. The measured speedup then briefly drops back below N. After 84 processors, at which time the speedup is over 92, the speedup remains superlinear.

Program 1 was expected to show a less significant difference between operating fully in cache and only partially in cache than Program 2, since the sequential memory access of Program 1 allows good use of cache (recall that each cache miss causes several following memory locations to also be cached). Ignoring context switches, moving completely into cache would improve the cache hit rate to about 100% from about 75%. This increases the effective speed of the processor, which we expected would be enough, considering the amount of communication in Program 1, to allow for superlinear speedup.

6.1.2 One Processor Per Node

As seen from Figure 5.4 and Table B.4, Program 1 achieved significant superlinear speedup when run utilizing one processor per node. In fact we began seeing superlinear speedup at only 42 processors, when the size of the local data set is still slightly larger than the size of the cache. By utilizing only one processor per node, we were able to see a speedup of around 221 times when utilizing 62 processors. The largest speedup we saw with Program 1 when utilizing two processors per node was 277, which occurred at 114 processors, so we certainly can go faster by utilizing the second CPU in each node, but we can conclude that given the choice to run Program 1 on either 62 dual processor nodes (124 processors total), or 124 single processor nodes, we would expect to see greater speedup with the single processor nodes, assuming the nodes have characteristics similar to the nodes on our test platform.

6.2 Program 2 Discussion

Like Program 1, Program 2 also showed superlinear speedup. As expected, the out of sequence memory access of Program 2 made poor use of the processor's cache until the local dataset fit entirely into cache. When the local dataset fits entirely into cache, at around 48 processors, after one iteration the data will all be cached until a context switch forces it out. That means that after one iteration of mostly cache misses (the constant used in the loop should be the only cache hit), we will have many iterations of all cache hits. Because of this we expected a spike in speedup after a the local dataset size passed a certain threshold.

An interesting observation that one makes when looking at Figure 5.2 is that although the local dataset is smaller than cache at 48 processors, we don't see a spike in speedup until we run on around 82 processors. This is also when the measured speedup of Program 1 was constantly above N. A possible explanation for this is the array that the program loops through is not the only data that we access from memory. For example,

each loop iteration we access a constant that we use in multiplication, a variable that we store the result of a multiplication and addition, and loop variables. These additional variables take up room in cache, and certainly would be a contributing factor to the delay between when the local dataset is smaller than cache and when we see a spike in speedup. However, at 82 processors the maximum local dataset is only 146,784 bytes, which is over 100,000 bytes smaller than the cache on our test platforms PIII processors.

6.3 Program 3 Discussion

Our major runs of Program 3 were set to communicate every 1,000 iterations. As we run on more processors the iteration time gets shorter, but we are communicating the same amount of data each time. That means that as we increase the number of processors, the communication gets more frequent and actually takes up a larger percentage of the total run time. We can qualitatively see in Figure 5.3 that the rate at which speedup is increasing as we increase N slows down somewhere between 40 and 66 processors. The speedup by moving completely into cache begins to offset and eventually overtake the additional communication time, and we see superlinear speedup, which is clearly observable in Figure 5.3. Note that the additional communication does lower the speedup that we see, especially for the larger processor counts, when compared to Program 2.

6.3.1 Communication Interval Tests

After analyzing the results from Table B.3 we decided to do limited runs of Program 3 at different communication intervals. First we set the communication interval to every 10 iterations and got the results listed in Table 6.1. The speedup observed in these tests has been plotted in Figure 6.1. As one can see this communication rate severely limited the speedup, keeping speedup under 14 as we ran on processor counts up to 124.

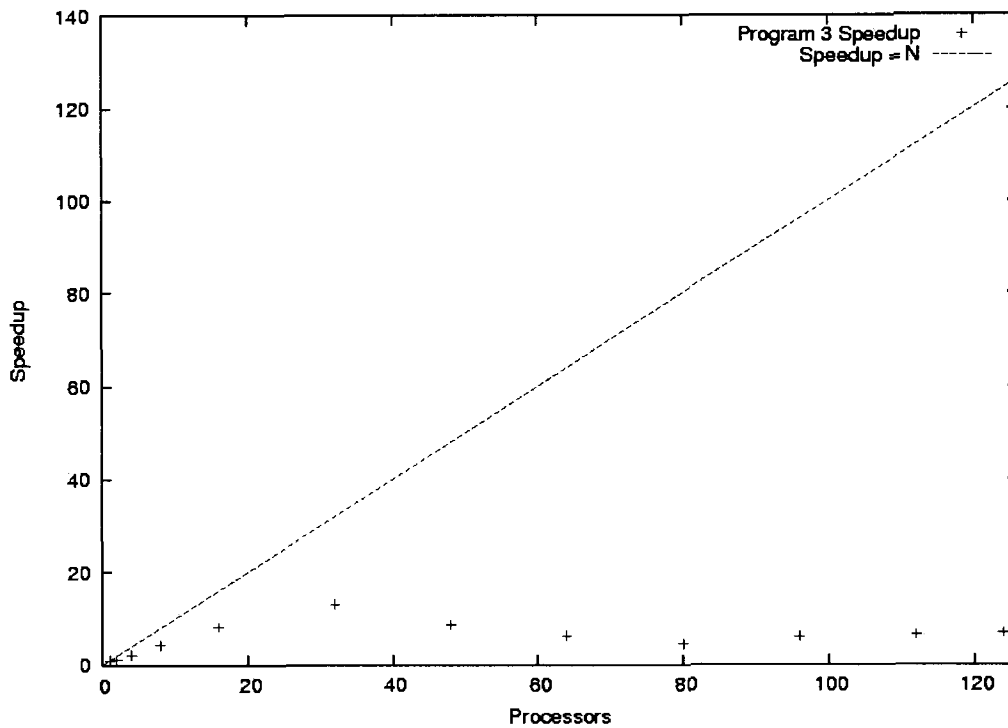

Figure 6.1: Program 3 Results, Comm Interval = 10 Iterations

Table 6.1: Program 3 Results, Comm Interval = 10 Iterations

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	81373.61	80835.84	81692.96	81300.81	1.08
4	3000000	40192.94	40700.28	39392.51	40095.24	2.18
8	1500000	19940.60	19970.57	20220.54	20043.90	4.37
16	750000	11134.34	10436.75	10558.58	10709.89	8.17
32	375000	7444.41	7177.85	5494.05	6705.43	13.05
48	250000	10821.03	11428.72	8091.05	10113.60	8.65
64	187752	14491.43	13396.12	14440.20	14109.25	6.20
80	150000	18676.87	19733.97	19859.72	19423.52	4.51
96	125000	16363.29	13786.79	13471.23	14540.43	6.02
112	107904	14610.50	15127.23	11328.24	13688.66	6.39
124	97536	12855.31	13088.36	12526.17	12823.28	6.82

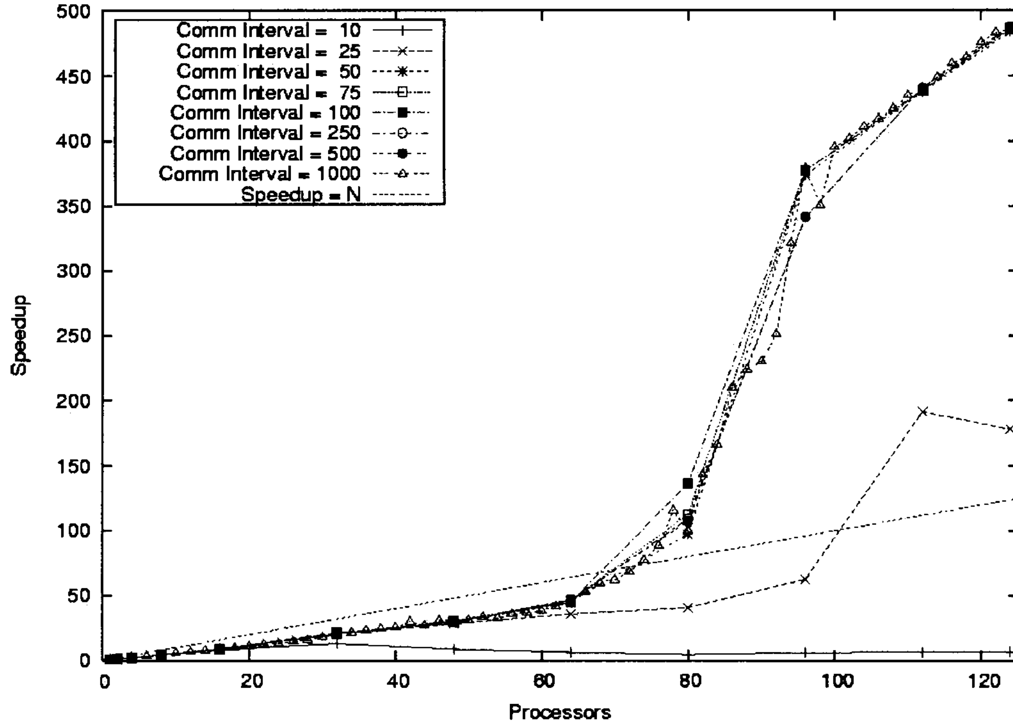

Figure 6.2: Summary of Additional Program 3 Tests

From the data shown in Table 6.1, we also decided to perform additional runs with communication intervals of 25, 50, 75, 100, 250, and 500 iterations. These results can be seen in Tables B.5, B.6, B.7, B.8, B.9, and B.10, located in Appendix B. Figure 6.2 compares the observed speedup in all additional Program 3 tests performed in Section 6.3.1, and the original Program 3 tests (as seen in Table B.3). All additional Program 3 tests were run utilizing two processors per node.

6.3.2 Communication Interval Test Discussion

One interesting thing to note in the additional Program 3 tests, was how quickly the speedup increased as the communication interval decreased. At a communication interval of 25 iterations, we saw superlinear speedup, although it required more processors to achieve it than at less frequent communication intervals. This is interesting because at a communication interval of 10 iterations the highest speedup we

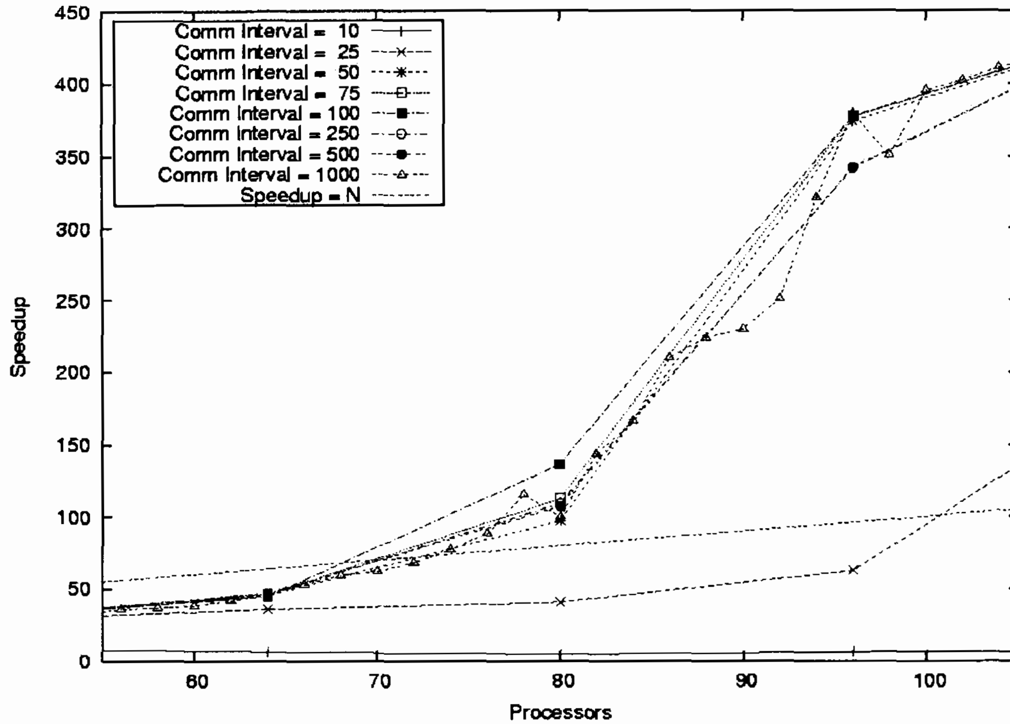

Figure 6.3: Additional Program 3 Tests, Detailed View

saw was a little over 13 when utilizing 32 processors. At a communication interval of 50 iterations the speedup was virtually identical to our original Program 3 tests with a communication interval of 1,000 iterations, as seen in Figure 6.2.

While typical CFD programs, such as CRAFT, communicate every iteration it is more important to consider the ratio of communication to computation rather than the frequency of communication. Even though CRAFT communicates once per iteration, the ratio of computation to communication is such that the speedup of the iteration increases almost linearly as the number of processors used increases linearly. Clearly our iterations are so short that communicating every 10 iterations severely limits the speedup, especially as the iteration time shrinks and the communication time remains the same or grows.

6.4 Multiple Array Tests

As seen in Figure 6.4, both of our multi-array test programs showed significant superlinear speedup. In MultiArrayTest 1, we accessed both arrays in the same loop, stepping through both of them out of sequence. Because of the way these arrays are accessed, we did not expect to see superlinear speedup until the local portions of each of the arrays both fit into cache at the same time.

In MultiArrayTest2 we basically broke our computation loop into two separate loops, where each loop iterated over a different array. Since we are only accessing one array at a time in this example, we would expect to see superlinear speedup earlier that we do in MultiArrayTest1. Since we did very limited (one run at 1,8,32,48,64,80,96, and 112 processors, compared to three runs starting at 1 processor and then every even processor count up to 124 for our three major test programs) tests with these two programs we will not draw too many conclusions from the results, other than in both cases we saw significant superlinear speedups.

6.5 Chapter Summary

From our test programs we have demonstrated that superlinear speedup due to microprocessor architecture is possible, because of the higher speed of on-chip cache memory. Furthermore, not only can superlinear speedup occur in programs that make poor use of cache, but is is even possible in programs that use the cache well. In the past, superlinear speedup was often attributed to inefficient sequential algorithms, but our examples demonstrated a remarkable speedup without using an inefficient sequential algorithm.

As expected we demonstrated an even larger speedup with a program that accessed memory out of sequence, as the benefits from cache were virtually zero when

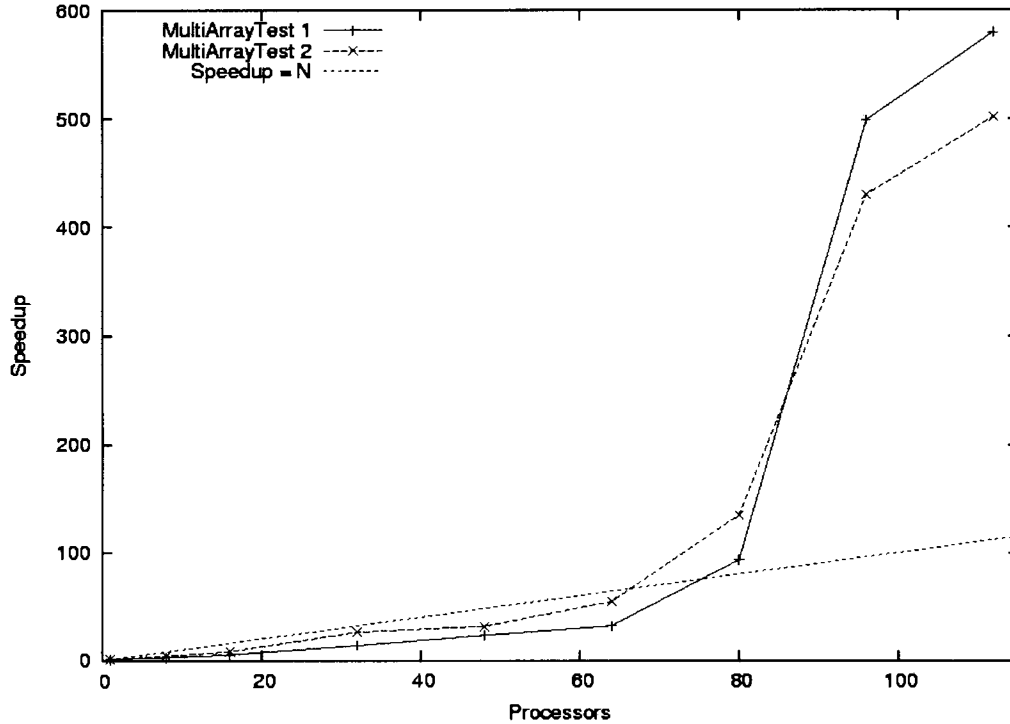

Figure 6.4: Multiple Array Test Results

the code was run sequentially, but once the local data fit into cache we saw a large jump in the effective speed of the computer.

We also saw superlinear speedups for every communication interval in Program 3, other than every 10 iterations. For this we saw an upper limit of around 13 on speedup, which occurred when the problem was run using 32 processors. If we were to look at the upper limit in speedup of 13 in terms of Amdahl's Law, as shown in Equation 2.2 we could conclude that the serial percentage of the code was approximately 10.5 percent, or an S value of 0.105, which is prohibitively high, and is much larger than many real world problems.

In summary we have determined that one of the most important aspects in determining the possibility of superlinear speedup for a particular program is the serial percentage of the code, which may include communication time. While the memory access order does have an effect on the speedup, programs were still able to see

large superlinear speedups with both sequential and non-sequential memory access. Any further conclusions drawn from these experiment results will be reserved for the following and final chapter.

CHAPTER 7

Summary and Conclusion

7.1 Summary

In this work we have summarized many of the theories and ideas concerning speedup of parallel computations. We also showed that distributed memory supercomputers can show large scale superlinear speedup without using tricks such as inefficient sequential algorithms or artificial time limits, such as those described by [18]. The reason Amdahl's Law does not apply in all cases, and why we were able to show large scale superlinear speedups was because of the tiered memory architecture of computers, and specifically the on-chip cache memory available in our test platform.

We performed tests with three major test programs, one of which iterated over a large array, accessing the elements sequentially. The second major test program differed from the first program in that it no longer just accessed adjacent array elements. The third major test program was similar to the second program, but instead of communicating once, partial sums were periodically sent from all nodes to the node with MPI rank of zero. In Program 1 and Program 2 the partial sums were only sent to the node with MPI rank of zero after the computation loop was finished.

All three programs showed significant superlinear speedup, as seen in Figures 5.1, 5.2, and 5.3. As expected, the transition between sublinear and superlinear speedup in Program 2 and Program 3 was quite dramatic, as shown in Figures 5.2 and 5.3 because these programs see very little benefit to the on-chip L2 cache until the local problems can fit entirely into the cache.

We had also performed additional tests to look at the effects of memory contention, communication interval, and utilizing more than one array. Our tests to investigate the effects of memory contention consisted of running our Program 1 test

program utilizing one processor per node and two processors per node. Since our test platform was made up of Pentium III nodes, which shared one memory bus for both processors, each processor would be competing for this resource when running our memory-bandwidth intensive program. We showed that speedup grew much faster when we utilized one processor per node rather than two processors per node (when we compare runs of equal processor counts). It is likely that a cluster built from nodes that have an independent memory bus for each processor would see much less of a difference between a run of 64 processors on 32 nodes and 64 processors on 64 nodes. However, there is more going on in our test than just the elimination of memory contention. When running two processes that each want to utilize a processor 100%, we are going to have instances of these processes being preemptively swapped out of their running state in order to allow a system process to run. Since our single processor per node tests were conducted on dual processor systems, the second processor was usually free for any systems processes that needed to run, therefore reducing the need to swap our process out of it's running state (which not causes the program to take more wall clock time, but it can also flush some of its data out of cache).

7.2 Predicting Superlinear Speedup

Through the use of our test programs, we identified the following properties that when possessed by a particular program allow for the possibility of superlinear speedup when run in parallel on a distributed memory supercomputer.

First, the program must iterate many times over a large singe or multi-dimensional array. Second, the local datasets must shrink as the problem is run on more and more processors. This does not work for problems where it is functionality not data that is distributed between processors. Third, the serial percentage of the program must be small. This means that the program must not require a large amount of communication, synchronization, or parallel overhead when run in parallel. Finally,

we have an additional property, which is not a requirement but increases the likelihood of superlinear speedup: out of sequence memory access.

Of these properties, the first two are the easiest to identify. Many of the traditional computationally intensive applications utilize large array-based datasets. Examples include CFD and finite-element simulations. It is also trivial to determine if the local data size of your problem shrink as you run it on more processors. It is also easy to determine if your code accesses your data out of sequence (*ie.* the indicies of a loop increase or decrease by a value of more than one each iteration). The difficulty comes in determining the serial percentage of the code. It is very difficult to quantitatively determine the serial percentage of code of a given algorithm, which is one reason why using an equation such as Amdahl's Law is not widely used to *predict* speedup, but instead is used to *explain* speedup.

So while we have identified these properties, we do not have a quantitative method for determining if or when a program will show superlinear speedup. There are some things we can say for certain about superlinear speedup when caused by the on chip cache. We contend that absolute upper limit on speedup will be the following equation, where R is the ratio of memory access speeds:

$$Speedup \leq \frac{S+P}{S+P/(N*R)}. \tag{7.1}$$

Note that in our case, where the L2 cache is about 10 times faster than main memory, the above equation would place an absolute upper limit of $10 * N$ on the speedup we could see by running on N processors. Also it is important to note that this equation would also work when discussing a dataset that goes from fitting entirely into cache to entirely fitting into CPU registers. Equation 7.1 does not *predict* speedup, since there are many more factors at hand in actual speedup such as parallel overhead (additional code, communication latency, etc), and the computer is only R times faster

for memory-fetch operations. Instead Equation 7.1 defines an absolute upper limit to speedup.

7.3 Conclusions

The first, and obvious, conclusion one should make from our work is that not only is superlinear speedup possible, but it is possible at a large scale in distributed memory supercomputers, and it is possible even when memory accesses are largely sequential in nature. This discovery is certainly counter-intuitive considering that the already cache-friendly program must see increased performance large enough to overcome any additional parallel overhead.

Secondly we would like to conclude that Amdahl's Law should not be viewed as a law, since we have shown that it can be broken. Instead, we should classify a subset of parallel programs that do follow Amdahl's Law as having "Amdahl-like parallelism".

It is not our intention to fault Amdahl for is assumptions about computer architecture. In 1967, when Amdahl published his paper promoting the uni-processor approach, computers were much simpler than they are today. This was, in fact, several years before the first microprocessor was introduced by Intel in 1971, or the first microprocessor powered a general purpose computer arrived in 1974[2]. It was not until many years after the introduction of the microprocessor that on-chip cache memory became popular, however off-chip cache memory was used in many large scale computing systems of the late 1960's and early 1970's[25].

Third, while this has yet to be proven, we feel comfortable concluding that the small superlinear speedups we saw in our CRAFT benchmarks could very well have been "real" superlinear speedup caused by the tiered memory architecture of our distributed memory supercomputer. More tests will be necessary to determine the exact cause of the superlinear speedup for that particular CRAFT problem, but certainly now that we know more about superlinear speedup we are well prepared for such a task. It

is interesting to note that the CRAFT code exhibits all of the properties identified in section 7.2.

While the tests conducted in this work were computationally expensive, we felt that not many researchers would have the opportunity to conduct tests as extensive as these. First quick turnaround times for jobs was a necessity because of the large number of runs we were required to make. Secondly, researchers would need access to an affordable computational platform. Third, researchers would need a reasonably large number of processors available to them. A system that meets all of these requirements is certainly not something that is available to everyone, which presented us with a unique opportunity to explore the area of superlinear speedup on distributed memory cluster supercomputers.

7.4 Future Work

There is room for a great deal of future work in this field. Now that we have discovered properties that can help us identify cases where superlinear speedup is possible, we can now look for real-world algorithms with these properties and run them on a cluster computer with input parameters that we feel will lead to superlinear speedups. This means we will then have used this work to successfully predict superlinear speedup in some other real world program.

Specific to our work with the CRAFT code, more profiling can be done to determine the exact cause of the superlinear speedup previously observed. Another area that needs more investigation is the relationship between problem size, cache size, and superlinear speedup. As we hypothesized, and proved, superlinear speedup can occur when local datasets fit entirely into cache after the parallel problem is run on enough processors. However, with many of our test results, superlinear speedup did not occur until the local data size was considerably smaller than cache. In some cases the local problem size was 100 kilobytes smaller than our 256 kilobyte cache before superlinear

speedup occurred. When utilizing one processor per node, superlinear speedup actually began before the local data set fit entirely into cache, and this was a problem that should already see a large benefit to cache since it makes sequential memory accesses.

All tests in this work could be extended to larger processor counts to find where our speedup begins to level off. Qualitatively we could see that there was "no end in sight", that is we had yet to see a point of diminishing returns, with the exception to Program 3 when run with a high communication rate as shown in Figure 6.1.

Another area in which our test can be expanded is by varying the problem size to determine how our speedups change as we shrink or expand our problem size. Again given the computational expense of the tests performed for this work, this would be a major undertaking that would take many months to accomplish. Despite expenses in both time and computational resources, this undertaking would add to our understanding of the mechanics of superlinear speedup.

Finally, the future problem we are most interested in is applying this work to the decomposition of large CFD problems, specifically, to the CRAFT code. Conventional wisdom says that if you want to run your CFD calculation on N processors you should decompose the problem into N equal-sized subproblems, giving one to each processor. What might be better is to decompose the problem into $M * N$ *cache-sized* pieces, giving M to each processor. The reason this may work well with the CRAFT code is because during each iteration the local problem is stepped through in all three dimensions. In the CRAFT code these are called an *I sweep*, a *J sweep*, and a *K sweep*. If, after the I sweep, the subproblem currently being worked in is now totally in cache the J and K sweeps will be able to work completely from cache. If the subproblem is very large, by the time CRAFT performs the J sweep most of the problem will have been flushed from cache. The same holds for the K sweep. By splitting the problem into $M * N$ cache-sized pieces rather than N larger pieces, we could possibly maximize our use of cache.

REFERENCES

[1] Intel, "Intel Research - Moore's Law," http://www.intel.com/research/silicon/mooreslaw.htm, 2004.

[2] Intel, "Microprocessor quick reference guide," http://www.intel.com/pressroom/kits/quickref.htm, 2004.

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 114–117, April 1965.

[4] W. Stallings, *Computer Organization and Architecture*. Upper Saddle River, New Jersey 07458: Prentice Hall, 2000.

[5] T. Hauser, T. I. Mattox, R. P. LeBeau, H. G. Dietz, and P. G. Huang, "High-cost CFD on low-cost cluster," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, (Dallas, Texas), 2000.

[6] B. P. Lester, *The Art of Parallel Programming*. Englewood Cliffs, New Jersey 07632: Prentice Hall, 1993.

[7] G. Bell and J. Gray, "What's next in high-performance computing?," *Communications of the ACM*, vol. 45, pp. 91–95, February 2002.

[8] T. Sterling, "The scientific workstation of the future may be a pile of PCs," *Communications of the ACM*, vol. 32, pp. 11–12, September 1996.

[9] P. D. Palma, A. Wiborg, and A. Withers, "Super computing on a budget," *The Journal of Computing in Small Colleges*, vol. 17, pp. 71–77, December 2001.

[10] K. Castagnera, D. Cheng, R. Fatoohi, E. Hook, B. Kramer, C. Manning, J. Musch, C. Niggley, W. Saphir, D. Sheppard, M. Smith, I. Stockdale, S. Welch, R. Williams, and D. Yip, "NAS experiences with a prototype cluster of workstations," in *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pp. 410–419, ACM Press, 1994.

[11] W. Gropp, "Learning from the success of MPI," Tech. Rep. ANL/MCS-P903-0801, Mathematics and Computer Science Division, Argonne National Laboratory, 2001.

[12] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, (Washington, D.C.), pp. 483–485, 1967.

[13] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.

[14] R. G. Brown, "Maximizing Beowulf performance," in *Proceedings of the 4th Anual Linux Showcase & Conference*, (Atlanta, Georgia), pp. 329–340, October 2000.

[15] Y. Shi, "Reevaluating Amdahl's Law and Gustafson's Law," http://www.cis.temple.edu/* *shi/docs/amdahl/amdahl.html, 1996.

[16] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proceedings of the 1990 Conference on Supercomputing*, (New York, New York), pp. 324–333, 1990.

[17] M. Thapar and B. Delagi, "Cache coherence for large scale shared memory multiprocessors," in *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 155–160, ACM Press, 1990.

[18] J. L. Gustafson, "Fixed time, tiered memory, and superlinear speedup," in *Proceedings of the Fifth Distributed Memory Computing Conference*, October 1990.

[19] V. Faber, O. M. Lubeck, and A. B. White, Jr., "Superlinear speedup of an efficient sequential algorithm is not possible," *Parallel Computing*, vol. 3, pp. 259–260, July 1986.

[20] D. Parkinson, "Parallel efficiency can be greater than unity," *Parallel Computing*, vol. 3, pp. 261–262, July 1986.

[21] D. P. Helmbold and C. E. McDowell, "Modeling speedup(n) greater than n," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 250–256, April 1999.

[22] S. Gibson and R. J. Hubold, "A perceptually-driven parallel algorithm for efficient radiosity simulation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, pp. 220–235, July-September 2000.

[23] Intel, *Intel Pentium III Processor for the PGA370 Socket at 500MHz to 1.13Ghz*, 8 ed., 2001.

[24] J. B. Rothman and A. J. Smith, "The pool of subsectors cache design," in *Proceedings of the 13th international conference on Supercomputing*, pp. 31–42, ACM Press, 1999.

[25] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[26] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI.* Cambridge, MA: MIT Press, 2 ed., 1999.

APPENDIX A

Source Code

Program 1

```
#include <stdio.h>
#include "mpi.h"

#define MAXNUM 1500000
#define NITER  1750000

int main(int argc, char **argv)
{
   double t1, t2, mySum, total, constant;
   double *myData;
   MPI_Status status;
   int rank, size, i, j, k, n;

   /* initialize MPI, get rank, size, and time */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   t1 = MPI_Wtime();

   mySum = 0;
   constant = 2; /*constant used in computational loop*/

   /**** SETUP LOCAL DATASETS *****/

   /*first each proc finds the size of its local data*/
   if(rank == 0 && MAXNUM % size != 0)
      n = MAXNUM % size + MAXNUM / size;
   else
      n = MAXNUM / size;

   /* each proc allocates memory for its data */
   myData = (double*)malloc(n*sizeof(double));

   /* each proc fills its data */
   for(i = 0; i < n; i++){
      myData[i] = 1.5;
   }

   /**** END DATA SETUP ****/

   /**** BEGIN COMPUTATION ****/
   for(i = 0; i < NITER; i++){
      for(j = 0; j < n; j++){
```

```
            mySum += myData[j]*constant;
        }
    }

    /* CONSOLODATE ANSWER */
    MPI_Reduce(&mySum, &total, 1, MPI_DOUBLE, MPI_SUM,
               0, MPI_COMM_WORLD);


    t2 = MPI_Wtime();
    MPI_Finalize();

    if(rank == 0){
        printf("----------------------------");
        printf("----------------------------\n");
        printf("The total is: %f\n", total);
        printf("The time is:  %f\n", t2-t1);
        printf("Num Procs = %d, ", size);
        printf("Size of local dataset: %d / %d bytes\n",
               (MAXNUM / size)*sizeof(double),
               (MAXNUM/size + MAXNUM % size)*sizeof(double) );
        printf("NITER = %d\n", NITER);
        printf("----------------------------");
        printf("----------------------------\n");

    }

    return 0;
}
```

Program 2

```
#include <stdio.h>
#include "mpi.h"

#define MAXNUM 1500000
#define NITER  1500000

int main(int argc, char **argv)
{
    double t1, t2, mySum, total, constant;
    double *myData;
    MPI_Status status;
    int rank, size, i, j, k, n;

    /*initialize MPI, get rank, size, and time*/
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    t1 = MPI_Wtime();
```

```
 mySum = 0;
 constant = 2; /*constant used in computational loop*/

 /**** SETUP LOCAL DATASETS *****/

 /*first each proc finds the size of its local data*/
 if(rank == 0 && MAXNUM % size != 0)
    n = MAXNUM % size + MAXNUM / size;
 else
    n = MAXNUM / size;


 /*each proc allocates memory for its data*/
 myData = (double*)malloc(n*sizeof(double));

 /* each proc fills its data */
 for(i = 0; i < n; i++){
    myData[i] = 1.5;
 }

 /**** END DATA SETUP ****/

/*step through out of sequence*/
for(k = 0; k < NITER; k++){
    for(i = 0; i<=4; i++){
       j=i;
       while(j < n){
          mySum += myData[j]*constant;
          j+=5;
       }
    }
 }

 /* CONSOLODATE ANSWER */
 MPI_Reduce(&mySum, &total, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);

 t2 = MPI_Wtime();
 MPI_Finalize();

 if(rank == 0){
    printf("----------------------------");
    printf("----------------------------\n");
    printf("The total is: %f\n", total);
    printf("The time is:  %f\n", t2-t1);
    printf("Num Procs = %d, size");
    printf("Size of local dataset: %d / %d bytes\n",
           (MAXNUM / size)*sizeof(double),
           (MAXNUM/size + MAXNUM%size)*sizeof(double) );
    printf("NITER = %d\n", NITER);
    printf("----------------------------");
    printf("----------------------------\n");
```

```
    }

    return 0;
}
```

Program 3

```
#include <stdio.h>
#include "mpi.h"

#define MAXNUM 1500000
#define NITER  1500000
#define COMM_RATE 1000
int main(int argc, char **argv)
{
   double t1, t2, mySum, total, constant, subTotal;
   double *myData;
   MPI_Status status;
   int rank, size, i, j, k, n;

   /* initialize MPI, get rank, size, and time */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   t1 = MPI_Wtime();

   mySum = 0;
   total = 0;
   constant = 2; /* constant used in computational loop */

   /**** SETUP LOCAL DATASETS *****/

   /* first each proc finds the size of its local data */
   if(rank == 0 && MAXNUM % size != 0)
      n = MAXNUM % size + MAXNUM / size;
   else
      n = MAXNUM / size;


   /* each proc allocates memory for its data */
   myData = (double*)malloc(n*sizeof(double));

   /* each proc fills its data */
   for(i = 0; i < n; i++){
      myData[i] = 1.5;
   }

   /**** END DATA SETUP ****/

   /**** BEGIN COMPUTATION ****/
```

```
   for(k = 1; k <= NITER; k++){
      for(i = 0; i<=4; i++){
         j=i;
         while(j < n){
            mySum += myData[j]*constant;
            j+=5;
         }
      }
      /* do partial sum */
      if(k % COMM_RATE == 0 || k == NITER ){
         MPI_Reduce(&mySum, &subTotal, 1, MPI_DOUBLE,
                    MPI_SUM, 0, MPI_COMM_WORLD);
         mySum = 0;
         if(rank ==0){
            total += subTotal;
            subTotal = 0;
         }
      }
   }

   t2 = MPI_Wtime();
   MPI_Finalize();

   if(rank == 0){
      printf("-----------------------------------------");
      printf("-----------------\n");
      printf("The total is: %f\n", total);
      printf("The time is:  %f\n", t2-t1);
      printf("Num Procs = %d, ", size);
      printf("Size of local dataset: %d / %d bytes\n",
             (MAXNUM / size)*sizeof(double),
             (MAXNUM/size + MAXNUM%size)*sizeof(double) );
      printf("NITER = %d\n", NITER);
      printf("-----------------------------------------");
      printf("-----------------\n");
   }

   return 0;
}
```

Multi Array Test 1

```
#include <stdio.h>
#include "mpi.h"

#define MAXNUM  750000
#define NITER  1500000

int main(int argc, char **argv)
{
```

```
double t1, t2, mySum, total, constant;
double *myData, *myData2;
MPI_Status status;
int rank, size, i, j, k, n;

/*initialize MPI, get rank, size, and time*/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
t1 = MPI_Wtime();

mySum = 0;
constant = 2; /*constant used in computational loop*/

/**** SETUP LOCAL DATASETS *****/

/*first each proc finds the size of its local data*/
if(rank == 0 && MAXNUM % size != 0)
   n = MAXNUM % size + MAXNUM / size;
else
   n = MAXNUM / size;


/*each proc allocates memory for its data*/
myData = (double*)malloc(n*sizeof(double));
myData2 = (double*)malloc(n*sizeof(double));

/* each proc fills its data */
for(i = 0; i < n; i++){
   myData[i] = 1.5;
   myData2[i] = 2.5;
}

/**** END DATA SETUP ****/

/*step through out of sequence*/
for(k = 0; k < NITER; k++){
   for(i = 0; i<=4; i++){
      j=i;
      while(j < n){
         mySum += (myData[j]*constant + myData2[j]*constant);
         j+=5;
      }
   }
}

/* CONSOLODATE ANSWER */
MPI_Reduce(&mySum, &total, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);

t2 = MPI_Wtime();
MPI_Finalize();

if(rank == 0){
```

```
      printf("----------------------------");
      printf("----------------------------\n");
      printf("The total is: %f\n", total);
      printf("The time is:  %f\n", t2-t1);
      printf("Num Procs = %d, size");
      printf("Size of local dataset: %d / %d bytes\n",
             (MAXNUM / size)*sizeof(double),
             (MAXNUM/size + MAXNUM%size)*sizeof(double) );
      printf("NITER = %d\n", NITER);
      printf("----------------------------");
      printf("----------------------------\n");

   }

   return 0;
}
```

Multi Array Test 2

```
#include <stdio.h>
#include "mpi.h"

#define MAXNUM  750000
#define NITER  1500000

int main(int argc, char **argv)
{
   double t1, t2, mySum, total, constant;
   double *myData, *myData2;
   MPI_Status status;
   int rank, size, i, j, k, n;

   /*initialize MPI, get rank, size, and time*/
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   t1 = MPI_Wtime();

   mySum = 0;
   constant = 2; /*constant used in computational loop*/

   /**** SETUP LOCAL DATASETS *****/

   /*first each proc finds the size of its local data*/
   if(rank == 0 && MAXNUM % size != 0)
      n = MAXNUM % size + MAXNUM / size;
   else
      n = MAXNUM / size;
```

```
 /*each proc allocates memory for its data*/
 myData = (double*)malloc(n*sizeof(double));
 myData2 = (double*)malloc(n*sizeof(double));

 /* each proc fills its data */
 for(i = 0; i < n; i++){
    myData[i] = 1.5;
    myData2[i] = 2.5;
 }

 /**** END DATA SETUP ****/

/*step through out of sequence*/
for(k = 0; k < NITER; k++){
    for(i = 0; i<=4; i++){
       j=i;
       while(j < n){
          mySum += myData[j]*constant;
          j+=5;
       }
    }
 }

 for(k = 0; k < NITER; k++){
    for(i = 0; i<=4; i++){
       j=i;
       while(j < n){
          mySum += myData2[j]*constant;
          j+=5;
       }
    }
 }



 /* CONSOLODATE ANSWER */
 MPI_Reduce(&mySum, &total, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);

 t2 = MPI_Wtime();
 MPI_Finalize();

 if(rank == 0){
    printf("----------------------------");
    printf("----------------------------\n");
    printf("The total is: %f\n", total);
    printf("The time is:  %f\n", t2-t1);
    printf("Num Procs = %d, size");
    printf("Size of local dataset: %d / %d bytes\n",
           (MAXNUM / size)*sizeof(double),
           (MAXNUM/size + MAXNUM%size)*sizeof(double) );
    printf("NITER = %d\n", NITER);
    printf("----------------------------");
```

```
        printf("-----------------------------\n");

    }

    return 0;
}
```

APPENDIX B

Detailed Results

Two Processors Per Node

Table B.1: Program 1 Results, Two Processors Per Node

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	79639.80	79638.32	79653.83	79643.98	1.00
2	6000000	51074.74	52024.08	51060.26	51386.36	1.55
3	4000000	35905.40	34734.53	34859.65	35166.53	2.26
4	3000000	25975.45	26011.63	25980.39	25989.16	3.06
6	2000000	17034.79	17046.41	17525.95	17202.38	4.63
8	1500000	13415.29	13183.46	13027.34	13208.69	6.03
10	1200000	10197.61	10219.56	10226.79	10214.66	7.80
12	1000000	8670.07	8491.01	8510.42	8557.16	9.31
14	857232	7539.51	7688.98	7349.24	7525.91	10.58
16	750000	6655.41	6647.08	6550.20	6617.56	12.04
18	666712	5981.07	5985.48	5985.14	5983.90	13.31
20	600000	5204.04	5103.78	5328.99	5212.27	15.28
22	545592	4728.79	4682.12	4861.15	4757.35	16.74
24	500000	4364.87	4484.54	4315.13	4388.18	18.15
26	461600	3925.80	4023.26	3829.63	3926.23	20.29
28	428664	3845.33	3310.82	3845.54	3667.23	21.72
30	400000	3373.81	3413.81	3437.10	3408.24	23.37
32	375000	3094.26	3173.91	3191.74	3153.30	25.26
34	353112	2942.58	2788.12	2956.83	2895.84	27.50
36	333520	2640.59	2544.33	2628.57	2604.50	30.58
38	315992	2515.13	2485.82	2402.13	2467.69	32.27
40	300000	2322.90	2386.13	2359.29	2356.11	33.80
42	285808	2374.18	2160.22	2168.83	2234.41	35.64
44	273040	2266.54	2045.06	2268.92	2193.51	36.31
46	261120	2024.39	2041.90	2028.25	2031.51	39.20
48	250000	1744.68	1870.67	1911.17	1842.17	43.23
50	240000	1740.60	1786.19	1662.05	1729.62	46.05
52	230832	1767.27	1892.93	1739.97	1800.06	44.25
54	222552	1483.40	1846.07	1706.35	1678.60	47.45
56	214600	1492.14	1780.47	1542.33	1604.98	49.62

(Continued on next page)

Table B.1: (continued)

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
58	206928	1551.42	1324.54	1549.45	1475.14	53.99
60	200000	1218.34	1356.25	1272.93	1282.50	62.10
62	193816	1438.03	1547.82	1345.62	1443.82	55.16
64	187752	1503.38	1475.48	1362.02	1446.96	55.04
66	181960	1283.54	1506.17	1286.92	1358.87	58.61
68	176912	1440.56	1157.24	1372.52	1323.44	60.18
70	171744	1369.16	1393.82	1378.69	1380.56	57.69
72	166856	1159.11	1195.25	1308.88	1221.08	65.22
74	162320	1177.50	1098.29	1054.86	1110.22	71.74
76	158400	1088.34	903.46	905.65	965.81	82.46
78	154320	1233.67	794.91	1138.49	1055.69	75.44
80	150000	806.49	1088.06	826.28	906.94	87.82
82	146784	1110.71	1056.03	884.08	1016.94	78.32
84	142952	820.27	766.81	985.70	857.59	92.87
86	140120	792.54	789.83	790.00	790.79	100.71
88	136680	791.45	800.86	847.64	813.32	97.92
90	133808	669.68	777.68	815.94	754.44	105.57
92	130688	491.91	1032.87	770.67	765.15	104.09
94	127992	733.27	511.45	603.22	615.98	129.30
96	125000	714.23	710.57	706.35	710.38	112.11
98	122544	561.49	831.51	527.63	640.21	124.40
100	120000	492.20	477.66	751.41	573.76	138.81
102	118360	489.11	759.90	790.54	679.85	117.15
104	115448	480.80	504.29	209.20	398.10	200.06
106	114000	723.60	526.83	422.05	557.49	142.86
108	111872	427.29	525.50	456.57	469.79	169.53
110	109408	375.30	416.44	753.55	515.09	154.62
112	107904	391.47	434.54	391.70	405.90	196.21
114	106072	191.43	410.74	258.93	287.03	277.47
116	103480	466.23	453.85	404.34	441.47	180.41
118	102504	184.99	411.09	391.20	329.09	242.01
120	100000	180.87	483.46	364.40	342.91	232.26
122	98440	370.27	355.66	177.83	301.26	264.37
124	97536	176.61	358.27	353.05	295.97	269.09

Table B.2: Program 2 Results

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	79608.79	80179.98	80475.30	80088.02	1.09
4	3000000	39883.54	39869.57	39798.33	39850.48	2.20
6	2000000	26670.20	26500.89	26666.33	26612.47	3.29
8	1500000	19805.03	19904.19	19928.92	19879.38	4.40
10	1200000	16023.06	16204.57	15917.52	16048.38	5.45
12	1000000	13276.94	13273.99	13253.79	13268.24	6.60
14	857232	11513.93	11332.75	11562.87	11469.85	7.63
16	750000	10085.42	10074.06	9994.40	10051.29	8.71
18	666712	8917.48	8904.62	8854.19	8892.10	9.84
20	600000	7898.90	7972.14	7842.31	7904.45	11.07
22	545592	6903.49	7169.18	6961.58	7011.42	12.48
24	500000	6654.59	6184.46	6185.83	6341.62	13.80
26	461600	5653.82	5718.10	6017.66	5796.53	15.10
28	428664	5154.93	5011.98	4899.15	5022.02	17.43
30	400000	4499.46	4354.59	4489.28	4447.77	19.68
32	375000	4424.57	4444.27	4339.13	4402.65	19.88
34	353112	4129.56	4237.50	4008.90	4125.32	21.21
36	333520	3910.91	4107.92	3831.59	3950.14	22.15
38	315992	3376.27	3857.07	3300.69	3511.34	24.92
40	300000	3446.71	3379.54	3532.93	3453.06	25.34
42	285808	3066.98	3144.68	3068.99	3093.55	28.29
44	273040	2995.13	2965.21	2941.09	2967.14	29.49
46	261120	2807.77	2892.98	2532.63	2744.46	31.89
48	250000	2412.97	2253.98	2261.98	2309.64	37.89
50	240000	2267.47	2532.87	2317.07	2372.47	36.89
52	230832	2298.12	2086.48	2286.50	2223.70	39.35
54	222552	2159.35	1797.64	2305.36	2087.45	41.92
56	214600	1987.69	2230.48	2008.35	2075.51	42.16
58	206928	1772.84	1735.27	1659.38	1722.50	50.80
60	200000	1710.48	1417.85	2100.59	1742.97	50.21
62	193816	1299.23	1461.29	1389.06	1383.19	63.27
64	187752	1808.78	1635.12	1904.05	1782.65	49.09
66	181960	1551.82	1505.39	1562.51	1539.91	56.83
68	176912	1471.73	1726.34	1791.55	1663.21	52.62
70	171744	1600.86	1056.27	1294.39	1317.17	66.44
72	166856	1483.00	1320.14	1362.04	1388.39	63.03

(Continued on next page)

Table B.2: (continued)

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
74	162320	1211.19	1304.20	1168.81	1228.07	71.26
76	158400	1019.16	1153.11	1286.39	1152.89	75.91
78	154320	993.02	1389.87	1006.02	1129.63	77.47
80	150000	1169.75	1285.15	1015.15	1156.68	75.66
82	146784	334.17	360.78	506.48	400.48	218.52
84	142952	454.49	565.67	568.75	529.64	165.23
86	140120	473.80	376.15	455.94	435.30	201.03
88	136680	292.52	294.14	352.11	312.93	279.65
90	133808	296.64	350.79	332.85	326.76	267.81
92	130688	323.09	323.62	322.23	322.98	270.95
94	127992	209.28	268.22	209.73	229.08	382.01
96	125000	205.34	205.23	287.94	232.83	375.85
98	122544	201.34	267.40	200.63	223.12	392.21
100	120000	197.34	198.29	197.26	197.63	442.80
102	118360	193.56	193.56	193.82	193.65	451.91
104	115448	189.31	189.58	189.12	189.33	462.20
106	114000	186.32	186.56	186.35	186.41	469.44
108	111872	182.88	182.88	182.90	182.89	478.49
110	109408	178.85	178.98	178.93	178.92	489.11
112	107904	176.52	176.49	176.48	176.50	495.82
114	106072	173.37	173.46	173.54	173.46	504.51
116	103480	169.41	169.33	169.24	169.33	516.80
118	102504	167.69	167.59	167.79	167.69	521.86
120	100000	163.84	163.64	163.73	163.74	534.46
122	98440	160.96	160.97	161.04	160.99	543.57
124	97536	159.42	159.45	159.50	159.46	548.80

Table B.3: Program 3 Results

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	78540.26	79135.58	78833.68	78836.50	1.11
4	3000000	39427.09	39061.88	39187.60	39225.52	2.23
6	2000000	26074.18	25726.23	26171.68	25990.70	3.37
8	1500000	19578.53	20143.72	19926.37	19882.87	4.40
10	1200000	15738.27	15665.33	15437.15	15613.58	5.60
12	1000000	13338.09	13560.78	13380.89	13426.59	6.52
14	857232	11093.30	11360.64	11596.19	11350.05	7.71
16	750000	9887.28	10146.65	9781.86	9938.60	8.81
18	666712	8835.09	8997.28	9044.91	8959.09	9.77
20	600000	8005.18	8074.82	7969.30	8016.43	10.92
22	545592	6882.77	7364.91	7354.12	7200.60	12.15
24	500000	6696.76	6560.72	6708.23	6655.24	13.15
26	461600	6164.53	5847.44	6069.97	6027.31	14.52
28	428664	5597.54	5842.94	5152.05	5530.85	15.82
30	400000	4772.94	5086.38	4928.86	4929.39	17.75
32	375000	4324.92	3810.00	4550.96	4228.63	20.69
34	353112	4284.43	3702.27	4210.67	4065.79	21.52
36	333520	3821.17	3783.84	3718.70	3774.57	23.18
38	315992	3748.81	3407.25	3584.59	3580.22	24.44
40	300000	3551.00	3478.58	3330.19	3453.26	25.34
42	285808	2470.95	3016.41	3344.15	2943.84	29.73
44	273040	3311.84	3324.60	3101.27	3245.90	26.96
46	261120	2823.14	2740.86	2944.55	2836.18	30.85
48	250000	2809.44	3015.49	2809.76	2878.23	30.40
50	240000	2987.25	2753.35	2692.85	2811.15	31.13
52	230832	2411.83	2679.30	2776.55	2622.56	33.37
54	222552	2575.01	2768.74	2708.85	2684.20	32.60
56	214600	2461.04	2441.41	2460.34	2454.26	35.66
58	206928	2539.71	2035.34	2554.43	2376.49	36.82
60	200000	2265.15	2236.58	2364.20	2288.64	38.24
62	193816	1955.38	2103.21	2231.50	2096.70	41.74
64	187752	1947.04	1876.62	1948.46	1924.04	45.48
66	181960	1686.83	1685.19	1589.65	1653.89	52.91
68	176912	1444.57	1478.40	1496.35	1473.11	59.41
70	171744	1377.21	1426.18	1423.08	1408.83	62.12
72	166856	1405.08	1349.05	1098.57	1284.23	68.14

(Continued on next page)

Table B.3: (continued)

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
74	162320	1160.19	1177.95	1060.74	1132.96	77.24
76	158400	1049.49	817.09	1105.43	990.67	88.33
78	154320	901.88	975.12	396.82	757.94	115.46
80	150000	854.48	854.44	924.23	877.72	99.70
82	146784	488.80	787.11	556.32	610.74	143.28
84	142952	487.02	470.96	621.63	526.53	166.20
86	140120	519.41	366.11	366.26	417.26	209.73
88	136680	398.10	378.22	397.03	391.12	223.75
90	133808	380.29	384.74	377.94	380.99	229.69
92	130688	357.24	320.95	369.39	349.20	250.60
94	127992	346.89	235.48	235.64	272.67	320.94
96	125000	230.74	230.51	230.86	230.70	379.32
98	122544	225.89	226.01	297.00	249.64	350.55
100	120000	221.21	221.40	221.50	221.37	395.32
102	118360	217.59	217.63	217.69	217.63	402.10
104	115448	212.41	213.00	213.23	212.88	411.08
106	114000	209.69	209.61	210.03	209.78	417.15
108	111872	205.98	205.66	205.71	205.78	425.25
110	109408	201.12	201.18	201.22	201.18	434.99
112	107904	198.30	198.37	198.57	198.41	441.05
114	106072	194.95	194.99	195.08	195.01	448.76
116	103480	190.44	190.34	190.61	190.46	459.46
118	102504	188.44	188.35	188.52	188.44	464.39
120	100000	184.05	184.06	184.14	184.08	475.38
122	98440	181.13	181.05	181.15	181.11	483.19
124	97536	179.29	179.30	179.28	179.29	488.10

One Processor Per Node

Table B.4: Program 1 Results, One Processor Per Node

CPUs	Size of Local Data (KB, Max)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	79639.80	79638.32	79653.83	79643.98	1.00
2	6000000	39819.81	39821.82	39817.47	39819.70	2.00
4	3000000	19912.07	19912.55	19908.76	19911.13	4.00
6	2000000	13273.06	13273.33	13273.39	13273.26	6.00
8	1500000	9957.05	9954.98	9955.10	9955.71	8.00
10	1200000	7965.31	7965.14	7964.50	7964.98	10.00
12	1000000	6637.73	6638.18	6638.30	6638.07	12.00
14	857232	5689.49	5689.43	5688.66	5689.19	14.00
16	750000	4978.62	4978.62	4977.86	4978.37	16.00
18	666712	4424.42	4424.61	4425.17	4424.73	18.00
20	600000	3982.66	3982.65	3982.63	3982.65	20.00
22	545592	3620.79	3620.76	3620.89	3620.81	22.00
24	500000	3319.13	3318.62	3319.24	3319.00	24.00
26	461600	3063.96	3064.00	3063.68	3063.88	25.99
28	428664	2845.23	2845.12	2845.24	2845.20	27.99
30	400000	2655.52	2655.33	2655.57	2655.47	29.99
32	375000	2489.59	2489.48	2489.63	2489.57	31.99
34	353112	2343.80	2343.57	2343.43	2343.60	33.98
36	333520	2212.68	2213.55	2213.58	2213.27	35.98
38	315992	2097.41	2097.24	2097.60	2097.42	37.97
40	300000	1991.70	1991.72	1991.72	1991.71	39.99
42	285808	1549.03	1490.08	1490.70	1509.94	52.75
44	273040	973.98	971.78	971.70	972.49	81.90
46	261120	584.38	574.94	567.18	575.50	138.39
48	250000	459.03	458.13	458.70	458.62	173.66
50	240000	438.56	437.51	438.19	438.09	181.80
52	230832	416.69	416.56	416.54	416.60	191.18
54	222552	401.30	401.95	401.39	401.54	198.34
56	214600	386.96	386.93	445.73	406.54	195.91
58	206928	373.15	552.91	373.13	433.06	183.91
60	200000	360.91	360.68	361.63	361.07	220.58
62	193816	349.45	381.35	349.43	360.08	221.19

Communication Interval Tests

Table B.5: Program 3 Results, Comm Interval = 25 Iterations

CPUs	Size of Local Data (KB)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	78264.00	78333.15	77982.40	78193.19	1.12
4	3000000	39279.27	39311.68	39668.83	39419.92	2.22
8	1500000	19921.98	20396.61	19547.40	19955.33	4.39
16	750000	9988.71	9511.61	10181.34	9893.89	8.84
32	375000	4077.41	4290.59	4392.02	4253.34	20.57
48	250000	3079.08	3045.43	3168.96	3097.82	28.25
64	187752	2285.99	2358.35	2683.62	2442.65	35.83
80	150000	1613.09	2948.18	1902.19	2154.49	40.62
96	125000	970.45	1311.06	1914.29	1398.60	62.57
112	107904	406.12	384.39	580.13	456.88	191.54
124	97536	554.52	463.68	456.14	491.45	178.07

Table B.6: Program 3 Results, Comm Interval = 50 Iterations

CPUs	Size of Local Data(KB)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	77602.91	78517.46	77777.94	77966.11	1.12
4	3000000	38938.14	39174.10	38175.37	38762.54	2.26
8	1500000	20299.21	19671.09	19508.11	19826.14	4.41
16	750000	10050.92	10075.87	10076.77	10067.85	8.69
32	375000	4072.28	4536.44	4306.27	4305.00	20.33
48	250000	2940.39	3012.87	2949.06	2967.44	29.49
64	187752	1990.38	1756.86	2031.68	1926.31	45.43
80	150000	830.56	897.89	973.13	900.53	97.18
96	125000	234.97	232.24	234.68	233.96	374.04
112	107904	199.88	199.58	199.84	199.77	438.07
124	97536	180.72	181.51	180.51	180.92	483.71

Table B.7: Program 3 Results, Comm Interval = 75 Iterations

CPUs	Size of Local Data(KB)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	78584.43	78539.77	78570.19	78564.80	1.11
4	3000000	39570.86	39221.01	38777.88	39189.92	2.23
8	1500000	20378.54	19727.73	19622.65	19909.64	4.40
16	750000	10110.43	9758.16	10038.65	9969.08	8.78
32	375000	4340.42	4129.73	4297.14	4255.77	20.56
48	250000	2840.40	2871.35	2930.67	2880.80	30.38
64	187752	1873.65	1817.85	1997.07	1896.19	46.15
80	150000	768.55	753.61	810.51	777.56	112.54
96	125000	230.63	232.80	232.76	232.06	377.09
112	107904	199.55	198.87	199.58	199.33	439.01
124	97536	180.18	180.23	180.01	180.14	485.79

Table B.8: Program 3 Results, Comm Interval = 100 Iterations

CPUs	Size of Local Data(KB)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	78088.92	78231.67	76712.87	77677.82	1.13
4	3000000	39365.95	39079.47	39136.86	39194.09	2.23
8	1500000	19442.09	19647.97	19540.95	19543.67	4.48
16	750000	9875.15	9808.71	9839.30	9841.05	8.89
32	375000	4147.10	3903.05	4266.13	4105.43	21.32
48	250000	3034.22	2988.67	2854.65	2959.18	29.57
64	187752	1876.82	2020.86	1962.82	1953.50	44.80
80	150000	647.56	754.16	525.76	642.49	136.20
96	125000	232.45	231.26	231.81	231.84	377.46
112	107904	199.24	198.81	199.88	199.31	439.07
124	97536	179.87	179.86	179.87	179.87	486.53

Table B.9: Program 3 Results, Comm Interval = 250 Iterations

CPUs	Size of Local Data(KB)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	78500.36	77997.22	78614.28	78370.62	1.12
4	3000000	38967.08	39321.77	39240.47	39176.44	2.23
8	1500000	19477.93	19394.22	20342.47	19738.21	4.43
16	750000	9878.19	9742.86	9741.19	9787.41	8.94
32	375000	4429.59	3872.24	4528.97	4276.93	20.46
48	250000	2851.26	3015.71	2967.10	2944.69	29.72
64	187752	1979.50	1804.24	1945.75	1909.83	45.82
80	150000	753.19	776.81	875.35	801.78	109.14
96	125000	231.76	305.59	231.69	256.35	341.37
112	107904	198.43	198.46	198.49	198.46	440.94
124	97536	179.48	179.42	179.44	179.45	487.66

Table B.10: Program 3 Results, Comm Interval = 500 Iterations

CPUs	Size of Local Data(KB)	Run 1(s)	Run 2(s)	Run 3(s)	Average(s)	Speedup
1	12000000	87511.87	87504.29	87514.48	87510.21	1.00
2	6000000	78500.36	77997.22	78614.28	78370.62	1.12
4	3000000	38967.08	39321.77	39240.47	39176.44	2.23
8	1500000	19477.93	19394.22	20342.47	19738.21	4.43
16	750000	9878.19	9742.86	9741.19	9787.41	8.94
32	375000	4429.59	3872.24	4528.97	4276.93	20.46
48	250000	2851.26	3015.71	2967.10	2944.69	29.72
64	187752	1979.50	1804.24	1945.75	1909.83	45.82
80	150000	753.19	776.81	875.35	801.78	109.14
96	125000	231.76	305.59	231.69	256.35	341.37
112	107904	198.43	198.46	198.49	198.46	440.94
124	97536	179.48	179.42	179.44	179.45	487.66

APPENDIX C

System Descriptions

Background Information

Blackbear is the primary supercomputer used by the University of Maine SDMT research group. Blackbear is based on 208 dual processor Pentium 3 1Ghz diskless compute nodes. This cluster is used for sensitive research, and therefore access is limited to researchers affiliated with the SDMT research project. In order to allow other University of Maine researchers to take advantage of computing resources that were often idle, but to still maintain the security of Blackbear, the Kearney cluster was created.

Kearney is comprised of 63 of the dual PIII compute nodes mentioned above, plus its own dual Xeon 2.8Ghz master node. These 63 nodes are physically separated from Blackbear by disconnecting appropriate Ethernet and Myrinet switch interconnects. The diskless nature of the compute nodes allows them to boot a different ram-disk image when connected to Kearney instead of Blackbear. Most of the time these 63 nodes are available to University of Maine researchers, however if the SDMT research group needs more than 145 compute nodes, the 63 Kearney nodes can quickly be reconnected to Blackbear. This reconnection process is quite simple: first, the Kearney master node is physically disconnected from the internal Ethernet and the Myrinet networks; second, the Ethernet and Myrinet switch interconnects are reestablished between the Kearney compute nodes and the Blackbear compute nodes; finally, the 63 Kearney nodes are rebooted to boot the ram-disk image for the Blackbear cluster.

Hardware

Blackbear

Master Node

- 2x Intel Xeon 2.8Ghz Processors
- 1024MB PC2100 RAM
- 2x 134GB Ultra-160 SCSI RAID0 Storage
- 1.4 Terabyte RAID5 Storage
- Intel Gigabit Ethernet - Management Network
- M3-PCI-64B Myrinet adapter - Computational Network

Diskless Compute Nodes (145 or 208)

- 2x Intel PIII 1.0Ghz Processors
- 512MB PC133 RAM
- Intel E100 Ethernet adapter - Management Network
- M3-PCI-64B Myrinet adapter - Computational Network

Kearney

Master Node

- 2x Intel Xeon 2.8Ghz Processors
- 1024MB PC2100 RAM
- 134GB Ultra-160 SCSI RAID0 Storage

- Intel Gigabit Ethernet - Management Network
- M3-PCI-64B Myrinet adapter - Computational Network

Diskless Compute Nodes (63)

- 2x Intel PIII 1.0Ghz Processors
- 512MB PC133 RAM
- Intel E100 Ethernet adapter - Management Network
- M3-PCI-64B Myrinet adapter - Computational Network

BIOGRAPHY OF THE AUTHOR

Glen Beane was born in Skowhegan, Maine on November 22, 1979. He graduated first in his class from Upper Kennebec Valley Memorial High School in 1998.

He entered the University of Maine in the fall of 1998 and obtained his Bachelor of Science degree in Computer Science in May of 2002.

In September of 2002, he was enrolled for graduate study in Computer Science at the University of Maine and served as a Research Assistant, performing research for the Supercluster Distributed Memory Technology (SDMT) research group. His current research interests include cluster computing.

He is a member of Upsilon Pi Epsilon, a national honor society for the computing sciences, and the Association of Computing Machinery. He is a candidate for the Master of Science degree in Computer Science from The University of Maine in August, 2004.