2009-01-01

Analysis of Cluster Compute Nodes With Varying Memory Hierarchy Distributions

Jon Valentin Ramirez
University of Texas at El Paso, jonramirez37@gmail.com

ANALYSIS OF CLUSTER COMPUTE NODES WITH VARYING MEMORY HIERARCHY DISTRIBUTIONS

JON RAMIREZ

Department of Electrical and Computer Engineering

APPROVED:

_____
David H. Williams, Ph.D., Chair

_____
Patricia A. Nava, Ph.D.

_____
Rodrigo Romero, Ph.D.

_____
Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

To my family, friends, and system administrators everywhere. May the fire and passion for learning never be extinguished.

ANALYSIS OF COMPUTE CLUSTER NODES WITH VARYING MEMORY HIERARCHY DISTRIBUTIONS

BY

JON RAMIREZ, BSEE

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

August 2009

Acknowledgements

Earning my Master's degree at the University of Texas at El Paso has been a great learning experience and an overall privilege. The lessons and knowledge gained molded me into the engineer that I wanted to be when I first set my sights on college.

I would also like to point out that this thesis would not have been possible without the encouragement and support from my family and friends. Thanks to my parents, Jose and Phuong Ramirez, for being outstanding role models and instilling in me the passion to learn and the will to work hard and never give up. Thanks to my brother, Jeremy, for his support and snacks. Thanks to my best friend, Cathy, for her support and overall expertise.

I would like to thank Dr. David H. Williams for his guidance and encouragement, not only during this study as my thesis advisor, but from the moment I walked into the graduate program at UTEP. I would also like to thank Dr. Patricia Nava and Dr. Rodrigo Romero for taking the time from their busy schedules to participate as members of my committee.

Thanks to my fellow system administrators in the Unix Lab for their support throughout this study. Special thanks go to Nito for all his help with this thesis. Thanks to Damian and Gabe for all the helpful discussions. Thanks to Melissa for lending me her books. I would like to thank all the faculty and staff at the University of Texas at El Paso's Electrical and Computer Engineering Department.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0709438. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Network switches were provided by Cisco Systems, Inc.

Abstract

This thesis compares a Compute Node, a cluster compute node that can completely contain smaller processes within, but not larger ones, with a Memory Node, a cluster compute node that can fully contain large processes within main memory. The Compute Node and Memory Node are tested by executing matrix multiplication programs that range in size from small processes similar to jobs that can be run on a single workstation to large processes that would only fit entirely within a Memory Node. Memory was allocated using four types of memory allocation: static, external, dynamic, and automatic. This analysis provides a measure of the benefit obtained by using a Memory Node over a Compute Node for large processes as well as a measure of the penalties that add up when a process is forced to utilize the swap device continuously during execution.

The test programs executed showed the benefit of using a Memory Node over a Compute Node, with speedup values peaking at approximately 700 for the process sizes used. Additionally, the tests showed that when a Compute Node is forced to swap to the hard disk multiple times during execution, the majority of the execution time will be devoted to accessing the hard disk, not performing computations. Furthermore, the results showed that the method of allocating memory does not make a significant difference in execution times.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Large Program Execution

Departments around the University of Texas, El Paso, as well as departments at other universities around the world, run a wide variety of programs ranging from simple programs such as word processors and internet browsers to memory intensive applications such as image processing, complicated simulations, or modeling of the human body. To run most programs, there is usually no need to use more than one workstation. When an application requires more memory to run than a single workstation has, additional steps must be taken in order to run these programs.

Computer clusters are groups of computers connected together in a network. Each member of a computer cluster, called a compute node, can be used independently to run programs, or controlled by a leader, a front end node, to work with other compute nodes in the cluster. Users typing a text document or executing a C program, for example, would not need to use the network. If a program that requires more memory than a single compute node contains needs to be executed, the limits of the single compute node can be overcome by distributing the program among two or more compute nodes at the same time using protocols such as remote procedure calls (RPC) or Message Passing Interface (MPI). If necessary, compute nodes can borrow idle memory from neighbors, enabling a computer to execute much larger processes than it would be able to with its own resources [And95]. Dedicated distributed memory servers (DDMS) can also provide remote paging and shared memory should a compute node not have enough resources for the job [Ift93, Rom03].

1.2 Limitations with Distribution

Although the limits of a single workstation are eliminated by the use of computer clusters, there are still limitations to distribution. If a compute node were to borrow resources from an idle neighbor, then it can continue with its own execution. A disadvantage, however, is that the program using the remote memory would not have priority over any program running locally on the lending machine. If the lending compute node were to execute a program, and the remaining resources on that machine become insufficient, the lending compute node will preempt the shared process to dedicate its resources to the local process. The shared process will have to reallocate remote resources from somewhere else and cannot continue with its computations until it does so [And95, Lit88]. Another serious disadvantage is that communication over a network is substantially slower than over a bus in a single machine. Thus, transferring data from one computer to another for storage and/or computation may slow processing rather than speed it up.

Computer clusters can also use DDMS to supply compute nodes with resources should the need arise. Unlike with borrowing from neighboring compute nodes, the resources borrowed from the DDMS would not be taken away because of a locally run process. Previous research at the University of Texas, El Paso, shows that this kind of setup relies on the capacity of the network. When testing programs with high demands for remote memory, the network was taken to the point of saturation, and the test program experienced excessive timeouts [Rom03].

It is also important to note that the operating system for the front end node or the compute nodes themselves do not distribute programs automatically. It is up to the programmer to use protocols like RPCs or MPI when writing their programs, or to design their programs to access memory from a DDMS. This in itself adds another layer of complication to a program.

1.3 Comparison of Compute and Memory Nodes

If a compute node does not have enough memory to execute a large process, then a possible solution is a memory node. A memory node is defined as a single workstation that has a significantly higher amount of main memory than a compute node. The two types of nodes would contain the same processors, motherboards, type of hard disk, and operating system. The main difference between a compute node and a memory node would be the memory hierarchy. Because a memory node would have considerably more RAM, it would need a motherboard that would accommodate that. Other than this, there would be no differences between the two types of nodes.

Using a memory node can provide an advantage when executing large programs because its main memory would be large enough that distribution or utilizing remote memory would not be necessary. Therefore, programs that would have needed to be designed with network protocols before would be designed to run on one workstation, reducing their complexity. Furthermore, there would be no need to use the network, so network saturation by that program would be nonexistent, and saturation caused by other programs using the network would not matter. Furthermore, the data communication speed is much faster over the internal bus in comparison to over the internet. The question with memory nodes, however, is whether or not the memory node would act like a compute node when executing large programs.

1.4 Thesis Objective and Organization

This thesis sets out to compare the performance of compute nodes and memory nodes while running processes of varying sizes, from small processes that resemble jobs programmers can execute quickly from their desktop computers to programs that are so large that only a memory node can possibly contain them within main memory without having to swap to the hard disk during execution.

Chapter 2 gives an overview of the theory involved in program execution that is relevant to this study. Chapter 3 describes the set of test programs used to perform the comparison and the experimental setup in which these tests were carried out. Chapter 4 presents the relevant findings during experimentation and the conclusions drawn from them.

Chapter 2

Theory

This thesis compares two types of cluster nodes by analyzing different aspects of their performance such as differences in execution time and the usage of system resources such as main memory and the hard disk. This chapter introduces the theory and concepts that affect the execution of the test programs during this experimentation. A description of process structure and execution follows. Finally, the concept of virtual memory and the mechanics that implement it are described.

2.1 Definitions

First, it is necessary to define several terms and how each affects a program or set of programs executed on a computer. These terms will be explained by considering the case of a programmer running more than one program, with one program being a matrix multiplication program.

- Thread: A thread is an executable unit of code that executes sequentially in a program and can be interrupted and scheduled by the operating system. Two or more threads can also execute in parallel within one program [Tan08]. In this case, the thread is the code compiled from the C syntax written by the programmer.
- Process: A process is a task executed by the operating system that consists of one or more threads and their resources [Bac90]. In this example, the process for the matrix multiplication program contains the thread that will be executed sequentially, any memory, and open files such as files containing matrix elements.
- Program: A program is an executable file, compiled to machine language from a programming language's syntax [Bac90]. A program can consist of one or more threads. The program in this example is the executable file created by compiling the matrix multiplication program written earlier.
- Multi-threaded: When a process is multi-threaded, then two or more threads can execute concurrently. Work in a process can be divided into multiple threads and executed concurrently [Tan08]. In the matrix multiplication program, the work can be divided into two threads and each thread can do half the calculations.
- Concurrent Execution: When a multi-threaded process runs, it may seem like two or more threads are executing at the same time. This is not the case, however. Rather than

running in parallel, the threads take turns running sequentially with each thread receiving a short amount of time to execute before the CPU moves to another one. This creates the illusion that both threads are running at the same time. Threads take turns running on all available CPUs, so it is possible to have more than one thread running at one time if there is more than one CPU, but all threads run sequentially on these CPUs [Tan08].

- System Call: System calls are user-callable routines that provide an operating system service [Bac90]. In the matrix multiplication program, the programmer can have all the matrix values stored in a file and call the system calls open() and read() so the operating system can open the file containing the matrix elements and read the contents into an array being used within the program.
- Multiprogramming: A computer that is multiprogrammed can hold multiple jobs in memory [Tan08]. For example, if the programmer is running the matrix multiplication program, and that program needs to access I/O to access matrix values from the hard disk, then rather than linger in an idle state, the CPU can switch to another job that is ready for computations and can run until the matrix multiplication program is ready to run again.
- Multitasking: A computer that employs multitasking schedules jobs so that multiple jobs take turns running on the same CPU or set of CPUs. Multitasking can be implemented using different scheduling rules that dictate the time each task is allowed to use the CPU based on factors such as the time spent executing so far or the priority of the tasks involved [Tan08]. For example, the CPU can run the programmer's matrix multiplication program and a web browser at the same time by allocating time to both processes and switching back and forth between the two for execution after set intervals of time.

- Multiuser Computer: A multiuser computer is one in which more than one user can access the computer in parallel. Users can access the computer remotely using programs like Secure Shell [Tan08]. For instance, in a multiuser computer, one programmer can be running a matrix multiplication program. Meanwhile, another programmer can log in using a remote shell and also run his own programs. Both users' processes are scheduled by the operating system and all programs execute concurrently.

2.2 Process Execution

A process contains the threads it should execute and any data the process needs. Processes must be structured a certain way to keep track of instructions, variables, and memory. The specific memory allocation for a C program executing as a process is structured as shown in Figure 2.1. A process consists of four main parts: the text region, the data region, the heap, and the stack.

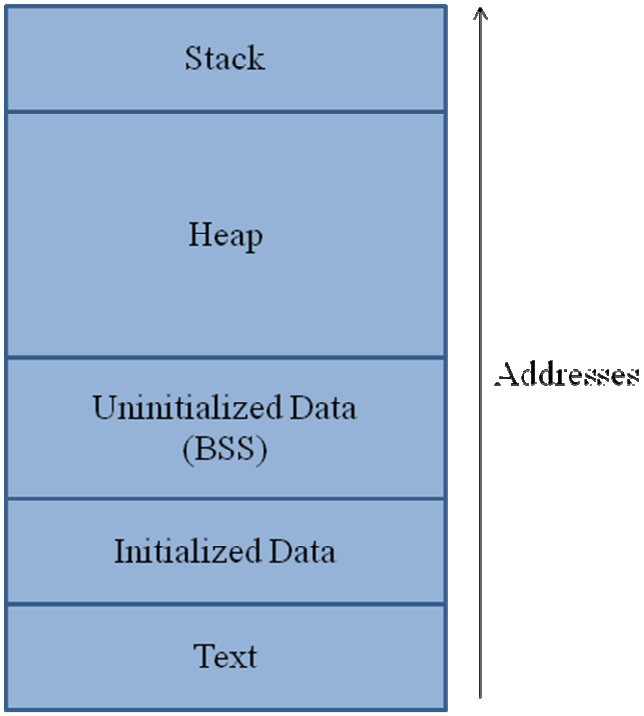

Figure 2.1 Layout of Memory for a C Program. Adapted from [Wil94].

Process instructions are stored in the text region. The text region is read-only because problems can arise if instructions stored in the text region are overwritten. Also, programs that modify their own code are not commonly used anymore [Tan08], so there is no need to have a

text region that is writable. Since no variables are stored in the text region, there is no need to change its size, so its size is constant.

The data segment consists of two main parts: the initialized data and the uninitialized data. The initialized data region contains variables that are given initial values. The uninitialized data region contains the variables that are not initialized, only storing the variable name and the length of the memory required for the variable. Two specific types of variables are stored in the data section. The first is static variables, variables that are only known in a single file and are initialized once before execution. After that, they retain their values if they are used again. The second type is external variables. External variables are available to functions within the same file or any file. External variables are defined as a form of global variable. Since it is necessary to change variables during program execution, the data region is writable.

Sometimes, variables are generated during execution using dynamic allocation. Such variables are placed in the heap, an area of free memory that can be reserved and freed during execution [Wil94]. The operating system accesses the heap by invoking the brk() system call to increase the size of the process [Bac90]. Programmers can also allocate memory from the heap by attaching it to a pointer using the malloc() function in C, which also calls brk(). The memory in the heap can be released using the free() function in C [Sch00].

The stack segment contains arguments, return addresses, automatic variables, and the processor state of the caller [Wil94]. The stack is different from the other memory segments in that rather than having a set size, the stack grows and shrinks as variables and data are pushed onto it and pulled from it, respectively. When a process starts, the stack contains any arguments coming from the command line or any function that called the working function as well as the return address of the calling function, so that the program can use them in execution and return.

A variable declared in a function in C without specifying either “static” or “extern” is an automatic variable. Automatic variables are pushed onto the stack and are only saved while the function they are declared within is executing.

If every process created executed sequentially and finished before the next process was allowed to start, then the CPU would be idle during slow operations such as read/write operations from the hard disk. Implementing multitasking and multiprogramming means that processes cannot always be executing their entire lifetime, processes that require access to the hard disk must yield execution to processes ready to run, and main memory must be reserved for processes ready to run. Figure 2.2 shows a model for the lifetime of a process as a state diagram [Bac90].

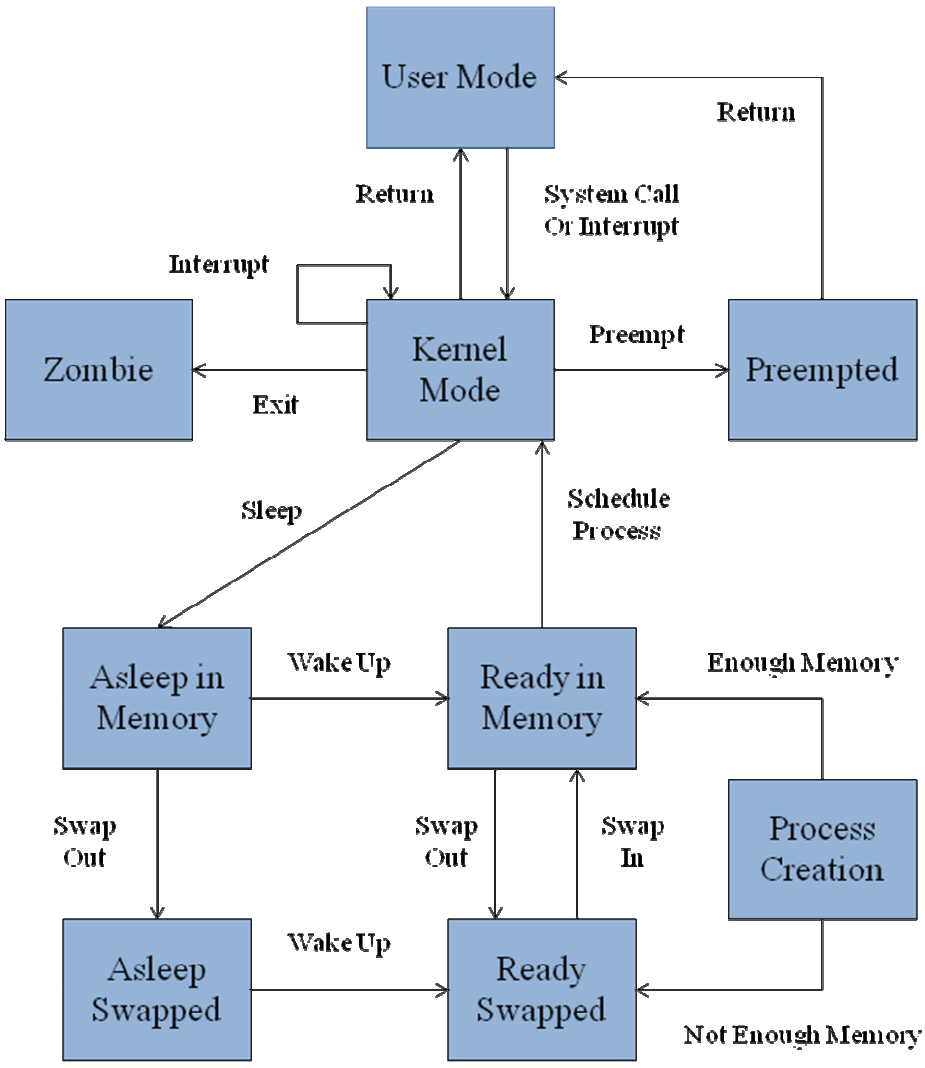

Figure 2.2 State Diagram for Process Lifetime. Adapted from [Bac90].

A new process enters the model at the Process Creation state. An existing process, called the parent process, invokes the fork() system call to create the new process, or child process. Once the child process is created, it moves to either the Ready Swapped or the Ready in Memory state. If there is enough room in main memory, the new process will go straight to the Ready in Memory state. Otherwise, it will move to the Ready Swapped state and remain there until it is

eventually swapped in by the swapper process. Then it will remain in the Ready in Memory state.

Once the process is in the Ready in Memory state, it must be scheduled for execution. When the process is scheduled, it moves into the Kernel Mode state. The Kernel Mode state is where a process resides if it is running in kernel mode. When the new process enters kernel mode for the first time, it completes its part of the fork() system call and moves to the User Mode state. In the User Mode state, the instructions stored in the process text region are executed. If the process encounters a system call, or an interrupt occurs, then the process moves back into the Kernel Mode state. There are several things that can happen once the process is back in the Kernel Mode state. The process can be preempted for another process, meaning the process will essentially be in the Ready in Memory state, except that rather than return to kernel mode when being rescheduled, it will return immediately to user mode [Bac90]. Interrupts are also handled completely within kernel mode.

When a process needs to access the hard disk, or perform some operation that causes the CPU to be idle, the process will go to sleep and enter the Asleep in Memory state. If another process requires resources, the process in the Asleep in Memory state may be swapped out to another device, causing it to enter the Asleep Swapped state. If the operation the process was waiting on finishes, then the process will wake up. The process will move from the Asleep in Memory State or the Asleep Swapped state to the Ready in Memory or Ready Swapped state, respectively.

In order for a process to terminate, it needs to execute the exit() system call, which causes the process to enter kernel mode and then the Zombie state. Once a process reaches the Zombie

State, it no longer exists, and it leaves records of its existence such as its exit code and timing statistics [Bac90].

2.3 Memory Management

If there were an infinite amount of RAM in a computer, then every process to ever run on that system could reside in main memory and only leave main memory upon execution of the exit() system call to enter the Zombie State. However, since there is a limited amount of RAM in any system, and it is unlikely that all processes can be held entirely within that memory, the CPU must be able to swap processes into the Swapped states as explained above. However, swapping processes in their entirety comes with disadvantages. One approach to solving the problems with swapping entire processes is to swap parts of a process. This section gives an overview of swapping and the disadvantages that come with it. Demand paged virtual memory is defined afterward with detailed explanations of paging, the swap device, and page faults.

2.3.1 Swapping

One way to make room for processes in main memory is to simply switch processes from the Asleep in Memory or Ready in Memory to the Asleep Swapped or Ready Swapped states, respectively, as shown in Figure 2.2. Bringing entire processes from the disk to main memory, running the process, and transferring it back to disk after a certain amount of time is called swapping [Tan08]. A process called the swapper swaps a process from main memory when the process needs memory. The swapper brings processes into main memory by analyzing the processes in the Ready Swapped State and choosing the one that was swapped out the longest [Bac90].

There are some disadvantages with swapping. If the swapper process swaps out a process, and the incoming process is larger than the process being swapped out, then another process will have to be swapped out. The second process being swapped out may have been larger than the incoming process making the first swap unnecessary. It is also possible that a process in the Ready in Memory State can be swapped out before it gets a chance to run again [Bac90].

2.3.2 Demand Page Virtual Memory

With virtual memory, each process executing has its own address space, and in most systems, that space is divided into equally sized pieces of memory called pages. When a program runs, it references its own addresses, the virtual addresses. When the CPU looks for the addresses, it accepts the virtual address referenced by the program as an input and translates that virtual address into the actual physical address on main memory, where it retrieves the data and continues operation. This translation is done through page tables in each region of the process. Each page has an entry in this table that contains the address in main memory where the physical page exists [Bac90]. With this method, the operating system gains three advantages. First, since chunks of memory entering and leaving main memory are all the same size, the problem of unnecessary swapping mentioned above is no longer an issue. Second, since the virtual addresses referenced by the programs are translated anyway, there is no need to keep the memory for a single process as a contiguous piece, meaning garbage collection to create larger chunks of free space is no longer necessary. Third, since each page is independent of the others in the process, there is no need for all pages to be in memory at once.

A computer that implements demand page virtual memory exploits the advantages gained by using a paged memory system. Not all pages in a process need to be in memory at one time for the process to run, so with demand page virtual memory, only the pages necessary for the process to currently run are loaded. Furthermore, if another process on the system needs resources, instead of evicting an entire process, if it possible, the operating system will only evict pages that have not been accessed for a long time from main memory, meaning one process' pages that are currently not being used can be removed with no consequence to that process, and the process requesting pages can receive the resources. The page table entries for demand page

virtual memory also contain information on how long the page has been in memory in addition to the physical address and valid bits already in the page table entries [Bac90].

If the operating system were to continue allocating pages to processes while keeping the processes in memory, eventually, pages would fill up main memory, and there would be no choice but to start swapping processes. A hybrid system that supports both demand paging and swapping can swap pages in whenever free memory is available and swap processes out whenever free memory is running low in a process called page reclamation [Bac90]. The operating system can decide which operation to use based on watermarks, thresholds on the amount of free memory in the system. Three thresholds are used: low watermark, minimum watermark, and high watermark [Som08]. As processes request pages and are serviced by the operating system, the number of free pages is decreased. When the number of pages free in main memory is above the high watermark, the system is balanced, and the system allocates pages to processes as they request them. Otherwise, the system is unbalanced. When the number of free pages crosses the high watermark, the system is considered unbalanced, but the operating system continues allocating pages. When the amount of free memory reaches the low watermark, however, the process swapper will wake up and begin page reclamation [Som08]. Pages are still being allocated to requesting processes at this time. When the amount of free memory reaches the minimum watermark, then pages are no longer allocated and any processes that request pages are placed in the Ready, Swapped State until the amount of free pages reaches the high watermark once again [Bac90]. Once it gets to this point, the swapper process goes to sleep and the operating system swaps pages again [Som08].

If all the pages for a process do not have to be within main memory at one time, then the event where a process needs but cannot reference a page is called a fault. When a fault occurs,

the operating system first sends the process into Kernel Mode and saves the program counter, general registers, and other information the process needs. Then the operating system determines the cause of the fault.

If the cause was an invalid virtual address from the program, then the process is killed [Tan08]. If the page still resides within main memory and has been designated as free to replace, then the page is reclaimed, and the fault is classified as a page reclaim. There is no need to access the hard disk to service a page reclaim [Bac90]. If the cause was an absent or invalid page, then the fault is classified as a page fault, and the operating system puts the process to sleep, sending it into the Asleep in Memory State while it completes the operations to bring the requested page into main memory from the hard disk [Tan08]. When a page needs to be evicted from main memory to make room for an incoming page, the page replacement policy used for demand paged virtual memory is the Least Recently Used (LRU) algorithm. When a new page needs to be sent into main memory, the page that has been in main memory the longest without being accessed is evicted [Abd05]. This method is appropriate since demand page virtual memory operates on the principle that pages that are being used at the time by a process should be in main memory while other pages not being used do not necessarily need to be there. Once the page is in memory, its page table entry is edited so that the operating system knows where it is when it receives its virtual address from the process. Then, the process wakes up, moves to the Ready in Memory State, then to Kernel Mode, and once the saved information from the first step is reloaded, back into User Mode [Tan08].

When main memory fills up, and processes need to migrate to main memory or pages are requested, the processes and pages being swapped out are put into the swap device, a section of the hard disk that is configured to hold data from processes in the Asleep, Swapped State and the

Ready, Swapped State as well as the pages from running processes that are not being used by the processes at that time [Bac90]. Using part of the hard disk as a swap device comes with a tradeoff between cost and performance. The main advantage of using a swap device as opposed to adding more RAM chips into the system is the cost. The hard disk space is cheaper than RAM chips, and a different memory hierarchy that uses the hard disk can handle processes as large as a system that would only use RAM cards. The drawback, however, is that access times from hard disk can be around six orders of magnitude slower than access times from the RAM cards [Abd05]. When a process needs to access the swap device repeatedly, the longer access times will stack up, leading to slower performance.

Chapter 3

Application

This chapter describes the experimental setup used to analyze the performance difference between the compute node and the memory node. It introduces the compute cluster used for the experiment with a description of the hardware and software used for all the compute cluster nodes and then compares the areas where the two types of nodes are different and explains why this is. Then the system calls and measurement tools used during this study are defined and explained. Finally, a detailed description of the experimental methods follows in three parts. The first part is a review of the matrix multiplication program and a justification for its use in this particular experiment. Then, the focus is on the details of the experiment: the test programs, compiling the test programs, and the structure for the experimental trials.

3.1 Hardware Setup

All test programs written for this thesis were executed on some of the compute and memory nodes for the cluster, Virgo 2, at the University of Texas, El Paso. Virgo 2 contains a front end node that connects to twenty-one compute nodes and two memory nodes. The front end and the other nodes in the cluster each contain two quad core Intel 5430 CPUs. The CPUs run at 2.66 GHz and contain 6.1 MB of cache space. Each node, including the front end also contains a DVD-ROM for software input as well as a 250 GB SATA hard drive. The front end node also contains four 500 GB hard drives which store user and application programs. The motherboards are the same with the exception of any hardware needed to access the RAM, which each node differs in order to be capable of containing their specified amount of RAM. The front end node and compute nodes each contain 8 GB of DDR2 RAM contained in 4 cards, while the memory nodes each contain 64 GB of DDR2 RAM contained in 16 cards.

The nodes are capable of executing processes that are smaller than the sum of main memory and the swap device [Ift93]. A memory node would be capable of running a 50 GB process without swapping because it fits entirely within main memory, for example. A compute node would also be capable of running the same process even with 8 GB for main memory only if its main memory and swap device exceeded 50 GB combined. The main difference, however, is in the memory hierarchy where the data used during execution is stored. If a program is larger than main memory, it can still run, but pages that are not currently being used have to be stored in the swap device until they are needed. The time taken to swap pages is a penalty because accessing the hard disk takes more time than accessing the RAM. This penalty is avoided if main memory is large enough to contain the entire process. The memory nodes were designed to contain entire processes. Having a much larger portion of the memory hierarchy consist of RAM enables large

programs to execute on a single machine without having to either execute on a single node while swapping, or execute on multiple nodes while sending large amounts of data between nodes.

3.2 Software Setup

The operating system and software applications are the same throughout the compute nodes and the memory nodes. The cluster uses ROCKS 5.0 to manage and monitor all nodes through one interface on the front end node accessible to operators. The operating system underneath ROCKS is the 64-bit Centos 5.0 Linux distribution, which uses the Linux 2.6.18 kernel. The operating system contains C, C++, FORTRAN, and JAVA compilers for software development as well as MATLAB. Programmers can also directly utilize the different cores and other compute nodes using threads, remote procedure calls, and message passing interface. The cluster has currently benchmarked at over 720 GFLOPS using the standard HPL benchmark.

3.3 Experimental Platform

A test program in this experiment must be easy to distribute and easy to vary in size yet in some cases be large enough so that a compute node will swap pages in order to execute the entire program. Furthermore, it must perform a large number of computations so that it is compute intensive. Matrix multiplication can use files potentially larger than a compute node's main memory as inputs, and each matrix element requires a series of computations, so it was chosen for this study. The test program reads in matrices of varying sizes from files, multiplies them together, and takes measurements.

3.3.1 Overview of Matrix Multiplication

The matrix multiplication done in this program follows the standard rules in linear algebra. Each element of the product matrix follows the equation:

Assume Matrix A with elements a_{ij} and dimensions MxN

Assume Matrix B with elements b_{ij} and dimensions NxP

Assume Product Matrix C with elements c_{ij} and dimensions MxP

i = rows and j = columns and N, M, and P > 1

Then, for all elements c_{ij} of Matrix C = AB

$c_{ij} = a_{i1}*b_{1j} + a_{i2}*b_{2j} + \ldots + a_{iN}*b_{Nj}$

The first reason why matrix multiplication was chosen for this study was the amount of memory two matrices can take up within memory. The program can be designed to multiply matrices large enough that the files used for the inputs can be larger than main memory for the compute nodes. Doing this can create the situation where using the LRU algorithm for demand page virtual memory will make the CPU repeatedly page values. Not only would the CPU need to page in each matrix element at least once, but if the programmer calculates matrix elements going left to right, row by row, the second matrix will have to cycle through values being paged in, and it is possible for each value in the second matrix to have to be paged in once for every row.

The second reason why matrix multiplication was chosen for this study was the number of calculations needed for this operation. The number of calculations can be found as follows:

Assume 2 matrices: MxN and NxP where N >> M, P and M, P > 1

Computations Per Element:

Additions: N-1 (Add all row and column products)

Multiplications: N (One multiply operation for each row/column element)

Total: 2N-1 calculations per element

Total Matrix Multiplication Calculations:

For MxN x NxP: $(2N-1)MP \approx 2NMP$

In all instances of the matrix multiplication program, M and P are set at 8 so that the product matrix would be an 8x8 matrix, a concise result that can be inspected quickly after experimentation. For example, in the 50 GB test case, the two matrices are 25 GB each. In order to obtain a 25 GB matrix with M and P at 8, N would have to be approximately $4x10^8$. Then, following the formulas above, the number of calculations would be approximately $5x10^{10}$.

Lastly, matrix multiplication was chosen for this study because of its versatility in the program's design. Only small changes are needed in order to change the program as needed during experimentation. The matrix sizes can be changed by changing two constants, and the type of memory allocation can be changed by changing just the initial variable declaration for the matrices. It does not take much work to manipulate the variables in the executables, and the resulting executables compiled from the changes are similar. Furthermore, matrix multiplication can be distributed easily by passing rows of the first matrix to the threads or compute nodes the job is being split between.

3.3.2 System Calls and Measurement Tools

This section lists the system calls, functions, and the measurement tools called in the test program and gives a brief overview of each function and why they are relevant to this experiment.

- malloc(): A C Language dynamic allocation function. malloc() allocates memory to the calling process from its heap during execution [Sch00].
- free(): A C Language function for dynamic allocation. free() works as an inverse to malloc(). free() takes the memory allocated by an earlier call to malloc() and releases that memory back to the system [Sch00].
- open(): A Unix system call. open() opens the file specified according to the flag it receives, making the file available for use. For this study, the flag, O_RDONLY, is used, meaning that the process is only permitted to read the file. open() returns a file descriptor that can be used for other system calls [Bac90].
- close(): A Unix system call. close() takes the file descriptor generated when open() was called earlier and closes the corresponding file [Bac90].
- read(): A Unix system call. read() takes the file descriptor generated by open() and reads the specified amount of data from the file, and places it into a buffer defined by the programmer [Bac90].
- gettimeofday(): A C callable function. gettimeofday() returns, in a structure called time_val, the amount of time that has passed since January 1, 1970. The time measure is given at microsecond precision as two long integers, one representing seconds and one representing microseconds. If gettimeofday() is called two times during a program,

an interval can be obtained for that time period by subtracting the values from the first call from the values from the second call.

- getrusage(): A C callable function. When called, getrusage() returns several pieces of information, such as time measurements and system statistics for either the calling process, or one of the child processes the calling process may have created. Like gettimeofday(), getrusage() measures time. However, the time obtained is only measured when the target process is running. When used in conjunction with gettimeofday(), it is possible to see how much time was spent on a certain process in relation to all the other processes running during a time interval. In addition, the time measurement is split into the time spent in user mode and the time spent in kernel mode. getrusage() also counts faults, which are relevant to this study since access to the hard disk is going to be observed.

3.3.3 Test Program Setup

The test program compares compute nodes and memory nodes by changing two variables: the size of the matrices being calculated and the memory region within the process where the matrix data are stored. The program runs the following steps:

- Allocate memory equal to the size of the matrices being multiplied together as well as the product at the end.
- Reads the matrices into the allocated matrices from files containing the generated matrices.
- Start the timing interval for the analysis tools.
- Perform matrix multiplication function
- End the timing interval for the analysis tools.
- Display the results

The way a variable is declared within a program is important because the declaration determines where the variable will be stored within the process model, as explained in Chapter 2. Therefore, the first step in the test program is to allocate the memory. Declaring the matrices as automatic variables places them in the stack, declaring them as static or external variables places them in the data region, and declaring the variables dynamically during program execution places them in the heap, extending the data region. Having the variables reside in all possible regions of the process model eliminates the possibility that the experimentation can be skewed because of the data allocation.

Once the memory for the matrices has been allocated, the program reads the matrices into that memory using the read() system call. The matrices being multiplied together in the program were generated separately using a random number generator. To have simplicity with the experimentation, two 25 GB matrix files were created to be used for the multiplication. The random numbers were set to between -100 and 100. This limit on the values was chosen so that there would be no chance of the matrix multiplication yielding a not-a-number (NaN) value for any of the matrix elements. All tests would read from the same files, and they would only read as much from the matrix file as needed. For example, if the test called for two 2 GB matrices, the program would only read the first 2 GB from each matrix file.

The matrix multiplication is done sequentially with one process. The main program calls the matrix multiplication function and passes the two matrix arrays. The product matrix is calculated one element at a time through loops and goes in order from the left to right, top to bottom. The product matrix is returned to the main program. gettimeofday() and getrusage() are called right before the matrix multiplication function is called as well as right after. Once the measurements have been taken, the total time returned from gettimeofday(), the getrusage() User Mode time, getrusage() System Mode time and getrusage() page fault count are calculated from the data.

The last step in the test program is to display the results. Obtaining an answer on the screen is not the important part of this step, because it serves as a check to verify that the matrix multiplication program ran correctly, and that any measurements taken by gettimeofday() and getrusage() are reliable. Previous matrix files that have been multiplied before in other applications were used in this matrix multiplication program, and the results displayed afterward verified the functionality of this matrix multiplication program. The measurements taken by

gettimeofday() and getrusage() are also output. The output on the display can be written to files so that data can be backed up.

The test program was compiled twenty-four times to yield twenty-four unique executables. The executables were broken up into four groups of six based on the memory allocation method. The four types of allocation are automatic, static, external, and dynamic. The matrix sizes vary within each group. The total size of the matrices for each executable are 4 GB, 8 GB, 12 GB, 20 GB, 30 GB, and 50 GB. The sizes start small so that the process size is small compared to the size of main memory for both compute nodes and memory nodes and gradually get larger to compare the results when the process size approaches the size of main memory for a compute node and when the process size is much larger than main memory for a compute node. The program was compiled through the command line with the GCC 4.1.2 compiler.

Executing programs that can reach sizes of 50 GB requires additional steps to work on both types of cluster nodes. When compiling a program without any arguments, the compiler requires that the program and its symbols be linked within the lower 2 GB of the address space. Because for many tests, the data region is already larger than 2 GB, the argument '-mcmodel=medium' is used so that only the program must be linked within the first 2 GB of the address space, eliminating the requirement for the symbols [Sta05]. Since the default stack limit for Centos 5.0 is 10240 MB, the stack limit had to be changed to accommodate the automatic allocation set of trials. This was done by adding an entry within /etc/security/limits.conf on each node granting the user a stack limit large enough to contain the largest matrices used.

Two compute nodes and two memory nodes are used for this experiment. Ten trials are done for each executable on each machine. There were some exceptions, however. For the compute nodes, not all ten scheduled trials took place because of time constraints due to the amount of

time each trial took for those machines, specifically for process sizes 20 GB and above. Two of each type of node were used to make sure that data collected were consistent for each machine.

Chapter 4

Results and Conclusions

This thesis set out to compare the performance of compute and memory nodes while running processes with a variety of sizes, from small processes that resemble jobs programmers can execute quickly from their desktop computers to programs that are so large that only a memory node can possibly contain them within main memory without having to swap to the hard disk during execution. Chapter 3 gave a description of the program that was written for this experiment and the experimental setup in which these tests would be carried out. This chapter presents the results obtained from this experiment and conclusions that were drawn from the data.

Since the results for all nodes were similar between the same types of nodes, only one compute node and one memory node were used for analysis. The results are presented in order of the type of data allocation used for the program: static, external, dynamic, and then automatic allocation. Within those sections, execution times are discussed followed by the fault data. The numbers on the charts and graphs shown in the following sections are the averages calculated from all the trials. Up to ten trials were executed for each data set, but it is important to note that for compute node results, the averages presented for the larger process sizes were not calculated from ten trials. This was because each of the large trials took several days to execute, so there was not enough time during this study to take the full ten trials for each type of memory allocation. The results from the two or three trials taken from each of these process sizes, however, were well within five percent of each other, so they were used. Lastly, those interested in the results of each individual trial are encouraged to look at the appendices where the raw data are labeled and located with the same organization as this chapter.

4.1 Static Allocation

First, the program that used static allocation for the matrix variables was used. Table 4.1 lists the averages for the total execution times for the matrix multiplication function, and Figure 4.1 shows the trends that these averages followed as the process size increased. As the process size increased, the execution time increased linearly for the memory node trials, while for the compute node trials, there was a drastic increase in execution time. The memory node showed significant speedups as the process size exceeded the size of the compute node's main memory.

Table 4.1 Static: Process Size vs Execution Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	169	1966	173666	216767	257331
Memory Node Time (s)	49	98	147	246	371	620

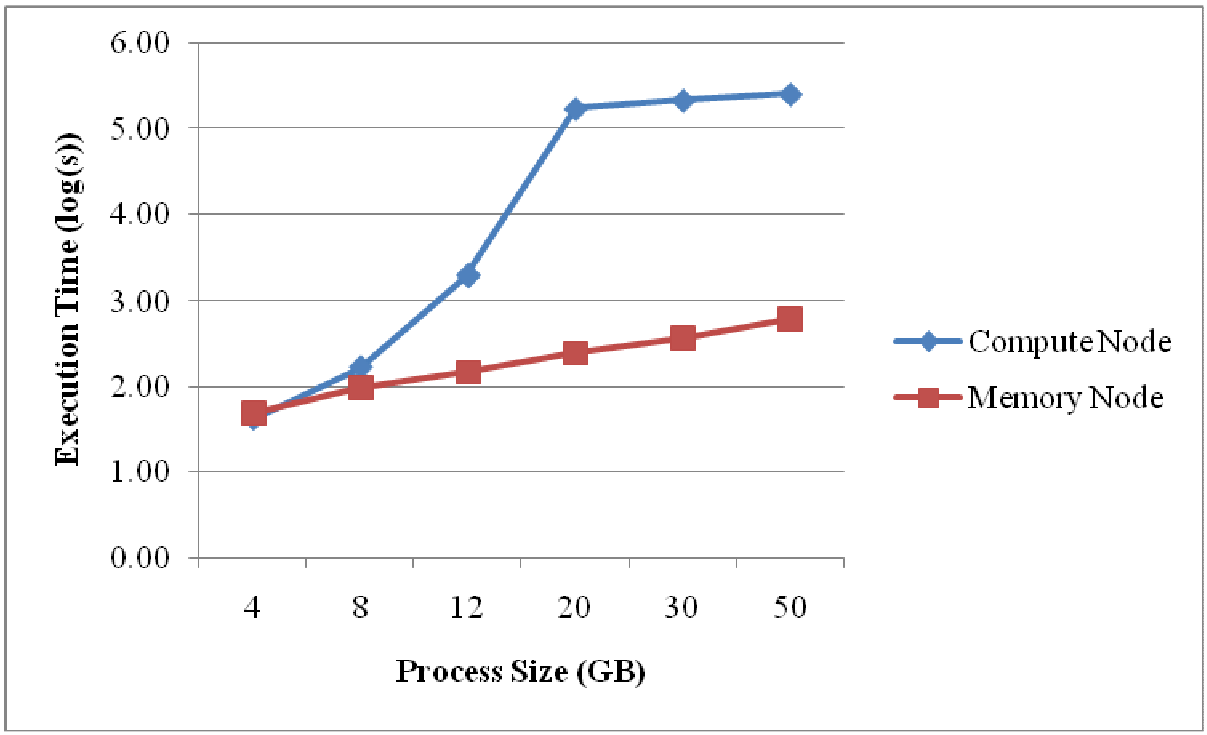

Figure 4.1 Static: Process Size vs Execution Time

The dramatic increase in execution time for the compute node occurred when the total process size first exceeded the size of the compute node's main memory. This was expected because once the process size exceeds the size of main memory, the operating system must swap pages to the swap device, and it can take up to six orders of magnitude more time for the operating system to fetch data from the hard disk as it does to fetch data from main memory [Tan08]. After the process size reaches 20 GB, the execution time still increases, but at a lower rate. This will be discussed further when page faults are explored. None of the process sizes were larger than the memory node's main memory, so there was no drastic increase in execution time. The largest speedup for this version of the program occurred at 20 GB, with a speedup of approximately 706, on average. At the largest process size measured, 50 GB, a speedup of approximately 415 was observed. This showed a clear advantage of using a memory node for large processes rather than a compute node.

Table 4.2 lists the averages for the amount of time the matrix multiplication function spent in user mode, and Figure 4.2 shows the trends that these averages followed as process sizes increased. These results show that there was not a large difference in the amount of time spent in user mode between the two types of nodes. Since program code is executed in user mode, these results indicate that the two types of nodes perform computations at approximately the same speed. It is also important to note a slight difference in the time between the memory node and the compute node. This can be attributed to the fact that because the memory nodes have a more complex RAM arrangement, where main memory is composed of sixteen cards rather than four, accessing RAM would take slightly longer, slightly increasing computation time for each instruction, thus increasing the time spent in user mode.

Table 4.2 Static: Process Size vs User Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	87	137	232	313	501
Memory Node Time (s)	49	98	147	246	370	620

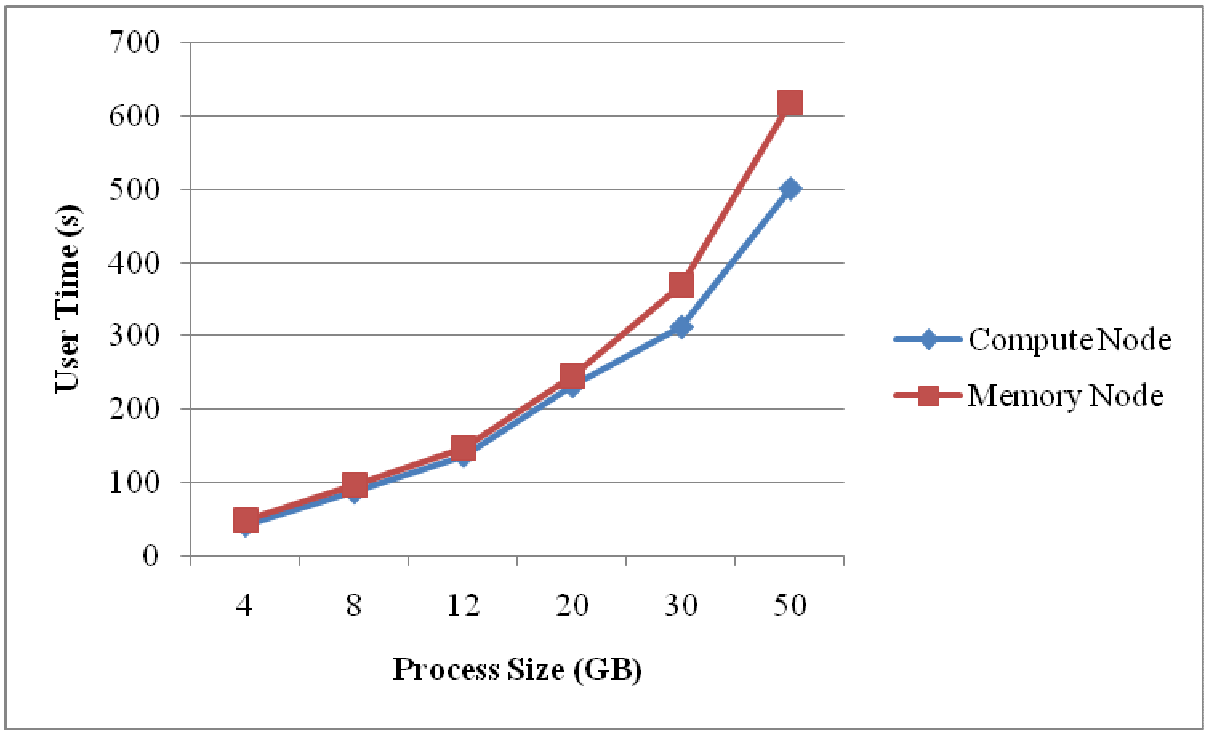

Figure 4.2 Static: Process Size vs User Time

Table 4.3 breaks down the averages for the amount of time the matrix multiplication function spent in kernel mode, and Figure 4.3 shows the trends that these averages took as the process size increased. Referring back to Chapter 2, when a process enters kernel mode, it does so because it either needs to wait for memory and goes to sleep, or there was an interrupt. In this case, the process would need to enter kernel mode for operating system interrupts and to fetch the matrix data it needs to execute when that data is not within main memory. Therefore, it was

expected that as the process size increased past the size of the compute node's main memory, the time spent in kernel mode would increase as well. It also follows that the memory node processes would not spend time in kernel mode to fetch data, and these results reflect that.

Table 4.3 Static: Process Size vs System Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	0	7	65	2222	3664	6093
Memory Node Time (s)	0	0	0	0	0	0

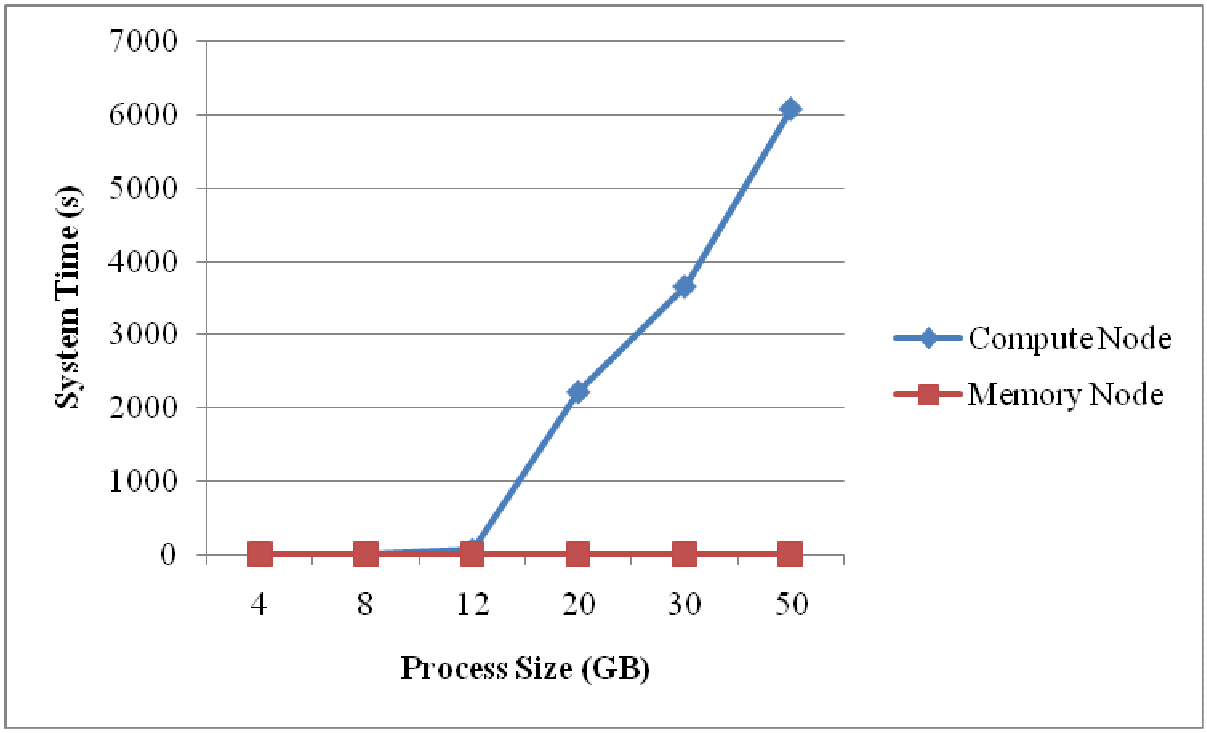

Figure 4.3 Static: Process Size vs System Time

Another major aspect in comparing the two types of nodes is the number of faults that occurred during execution. Table 4.4 shows the averages for total number of faults that occurred during execution, and Figure 4.4 illustrates the trends that these averages took as the process size

increased. The number of faults increased as the process size increased past the main memory size for the compute nodes, while no faults occurred during execution on the memory nodes. Since faults occur when data cannot be found, it was expected that faults would occur when trying to access a page that has been swapped out. Likewise, it can be expected that if the entire process can be stored on main memory, then nothing would be swapped out, and all pages would be accessible to the program from main memory. These results reflect this by showing that compute nodes, which do not have a large enough main memory to contain large processes, had an increasing number of faults occur during execution as process sizes increased, while memory nodes, which have a main memory large enough to contain all test programs, had no faults occur during any trial.

Table 4.4 Static: Process Size vs Total Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	450915	3084013	122070016	256816753	467692695
Memory Node	0	0	0	0	0	0

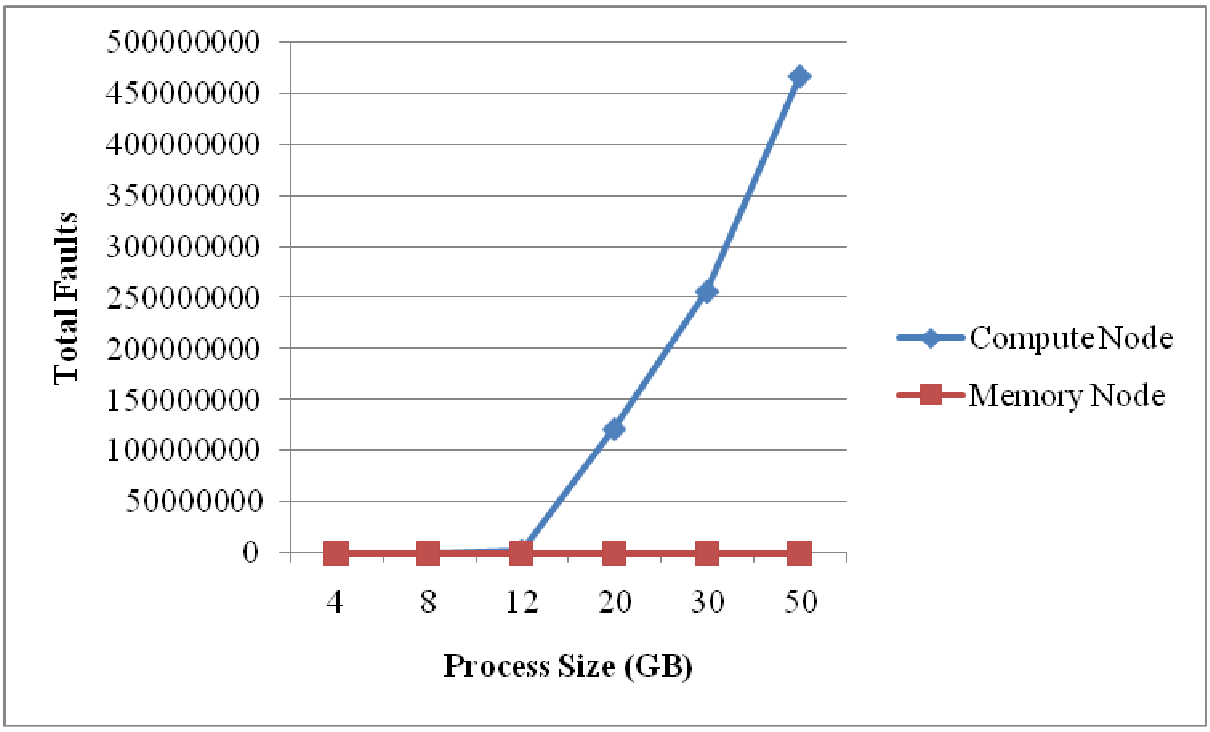

Figure 4.4 Static: Process Size vs Total Faults

Out of the total number of faults that occurred during execution, a percentage of them will be page faults, and the rest will be page reclaims. Referring back to Chapter 2, page faults require access to the swap device while page reclaims and do not. It is important to separate page faults from page reclaims since accessing the disk can take up to six orders of magnitude longer than accessing main memory. Table 4.5 shows the averages for the number of page faults that occurred during execution, and Figure 4.5 illustrates the trends that these averages took as the process size increased. At the same time, Table 4.6 shows the averages for the percentage of the total faults that are page faults, and Figure 4.6 illustrates the trend that they follow. The percentages only applied to the compute nodes since the memory nodes did not have any faults.

Table 4.5 Static: Process Size vs Page Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	59940	519939	35574612	54381025	80673173
Memory Node	0	0	0	0	0	0

Table 4.6 Static: Page Faults Percentage

Process Size (GB)	8	12	20	30	50
Percentage Page Faults	13.29	16.72	29.14	21.18	17.25

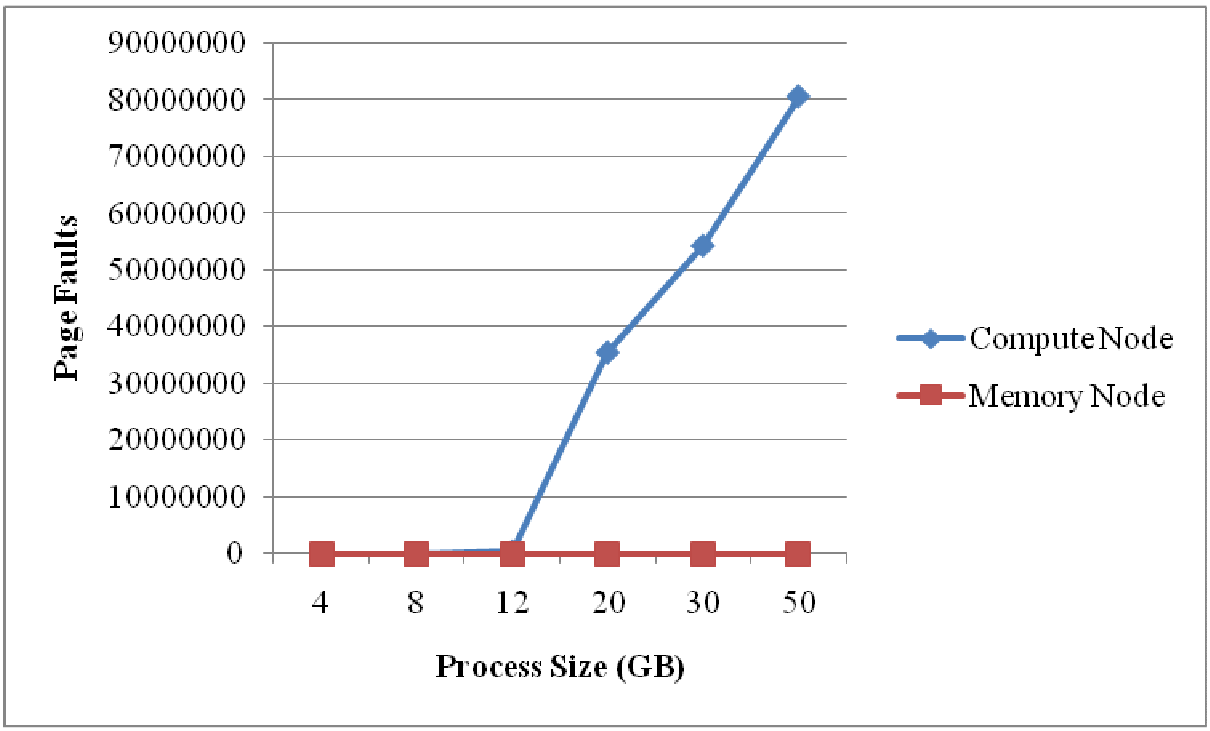

Figure 4.5 Static: Process Size vs Page Faults

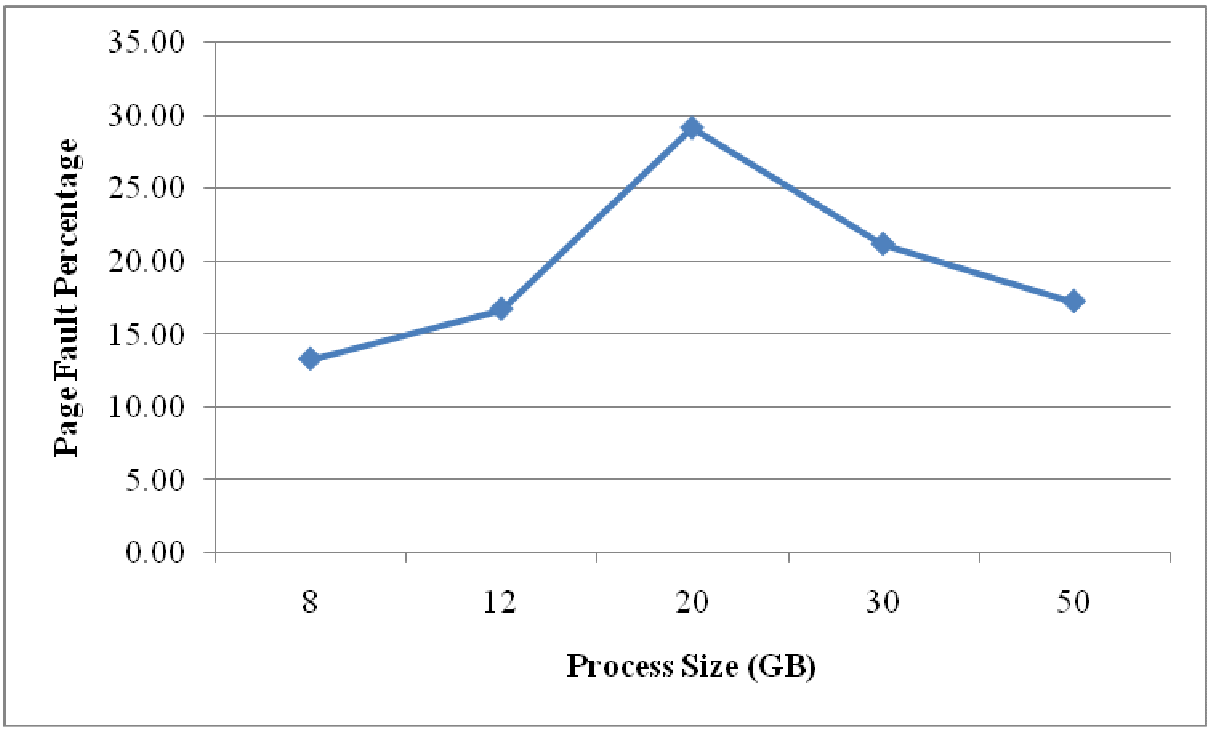

Figure 4.6 Static: Page Fault Percentage

As the process size increases, the number of faults increases, and therefore, the number of page faults increases, as verified by these data. However, the percentage of total faults that occurred that were page faults did not stay constant. The percentage peaked around the 20 GB process size at 29 percent. Afterward, the percentage decreased as the process size decreased. This is significant because a page fault, which requires disk access, can take up to six orders of magnitude longer to service than a page reclaim, which does not require disk access. Even though the total number of faults increases as the process size increases, if the page fault percentage decreases, the execution time will not increase as rapidly.

4.2 External Allocation

Now the version of the program that employed external allocation will be examined. There was not a significant variation in these results and the static allocation results because, with external allocation, the data was stored within the data region, just like with static allocation, not creating a significant difference in where the program accesses memory. Table 4.7 lists the averages for the total execution times for the matrix multiplication function, and Figure 4.7 shows the trends that the averages listed followed as the process size increased. The execution time increased linearly for the memory node as the process size increased. On the other hand, the compute node execution times did not increase steadily with the process size. There was a large increase in execution time starting around the point when the process became larger than main memory. The memory node showed significant speedups on average when compared to the compute node.

Table 4.7 External: Process Size vs Execution Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	167	2390	178307	219304	255372
Memory Node Time (s)	49	99	148	248	373	627

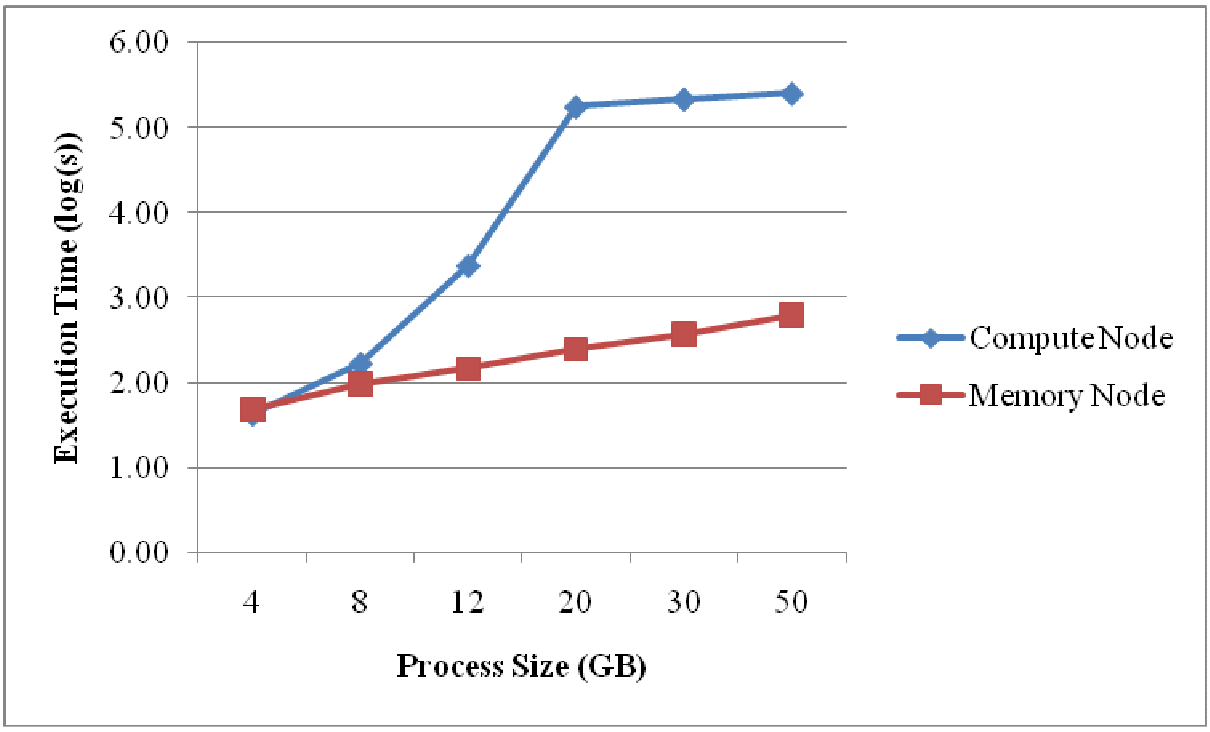

Figure 4.7 External: Process Size vs Execution Time

As with the static case, the dramatic increase in execution time for the compute node occurred when the total process size first exceeded the size of the compute node's main memory. This was expected because once the process size exceeds the size of main memory, the operating system must swap pages to the swap device, and it can take up to six orders of magnitude more time for the operating system to fetch data from the hard disk as it does to fetch data from main memory. After the process size reached 20 GB, the execution time still increased, but at a lower rate. None of the process sizes were larger than the memory node's main memory, so there was no drastic increase in execution time. The largest measured speedup occurred when the process size was 20 GB, where a speed up of approximately 719. At a process size of 50 GB, a speedup of approximately 407 was measured. This showed a clear advantage of using a memory node for large processes rather than a compute node.

Table 4.8 lists the averages for the amount of time the matrix multiplication function spent in user mode, and Figure 4.8 shows the trends that these averages followed as process sizes increased. These results show that there was not a large difference in the amount of time spent in user mode between the two types of nodes. Since program code is executed in user mode, these results indicate that the two types of nodes perform computations at approximately the same speed.

Table 4.8 External: Process Size vs User Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	87	137	232	313	494
Memory Node Time (s)	49	99	148	248	373	627

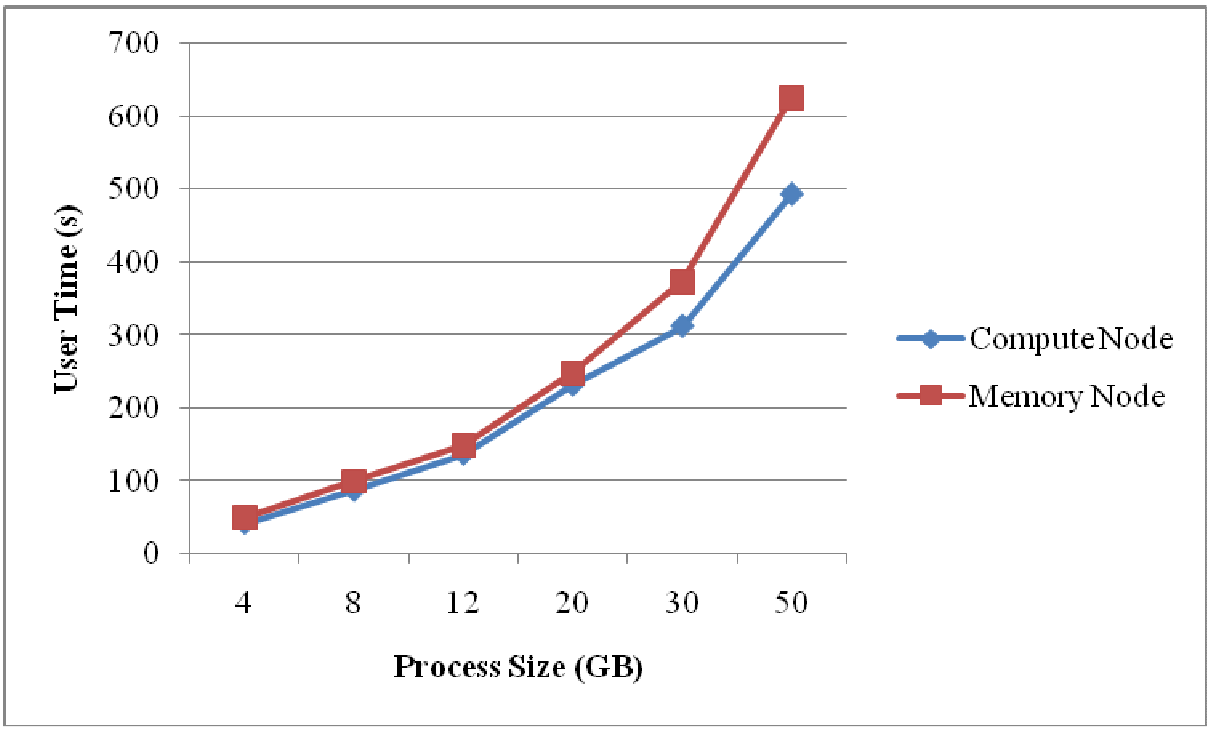

Figure 4.8 External: Process Size vs User Time

Table 4.9 lists the averages for the time that the matrix multiplication function spent in kernel mode, and Figure 4.9 shows the trends that these averages followed as process sizes increased. Like the static version of the program, because the process was larger than the compute node's main memory for some trials, it was necessary for the process to enter kernel mode at some point during execution. As the process size increased past the size of the compute node's main memory, the time spent in kernel mode increased as well. Since the memory node's main memory was larger than any process size used during this study, there was no need for its processes to enter kernel mode, and the results show that they did not.

Table 4.9 External: Process Size vs System Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	0	7	70	2228	3671	5909
Memory Node Time (s)	0	0	0	0	0	0

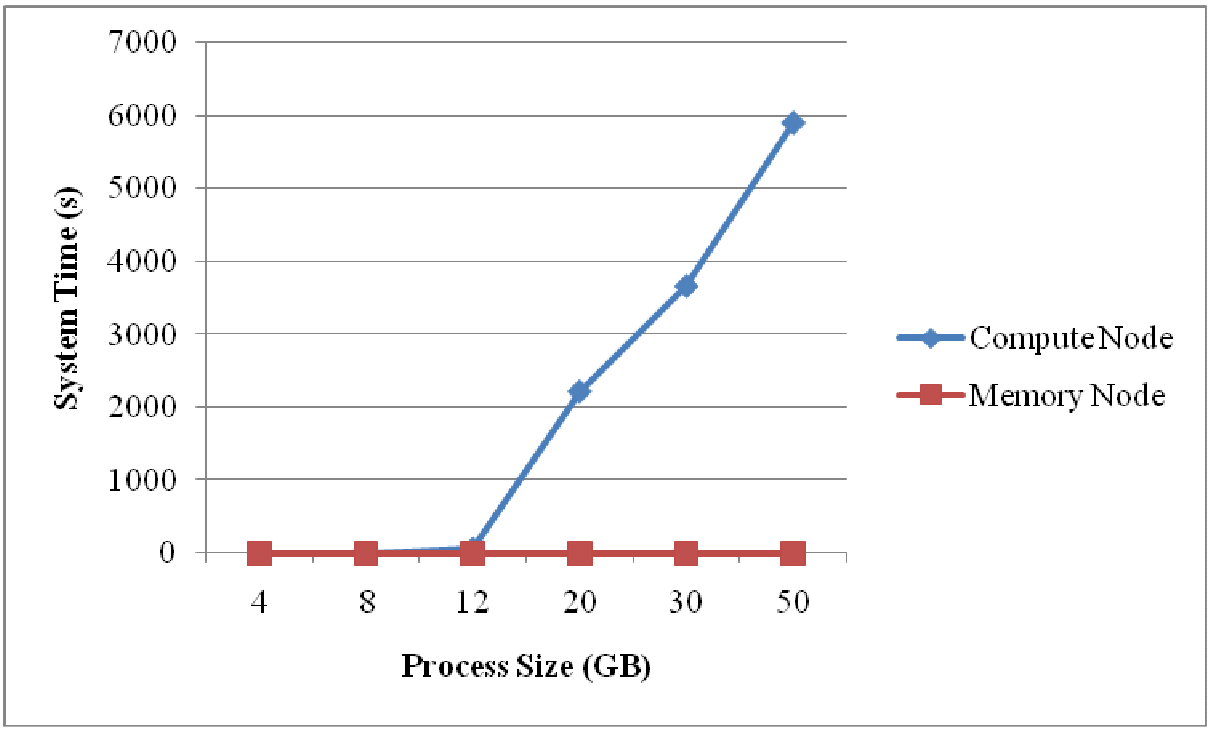

Figure 4.9 External: Process Size vs System Time

Another aspect important in comparing the two types of nodes is the number of faults that occurred during execution. Table 4.10 shows the averages for total number of faults that occurred during execution, and Figure 4.10 illustrates the trends that these averages took as the process size increased. On the compute node trials, as the process size increased after surpassing the size of the compute node's main memory, the number of faults that occurred during those trials increased. On the other hand, no faults occurred during the execution of the matrix multiplication function on trials executed on the memory node. These results reiterated that if the process size is larger than the size of main memory, then the entire process cannot fit within main memory, and when data cannot be found in main memory because of this, faults will occur.

Table 4.10 External: Process Size vs Total Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	455561	3245448	122261239	260613993	467677325
Memory Node	0	0	0	0	0	0

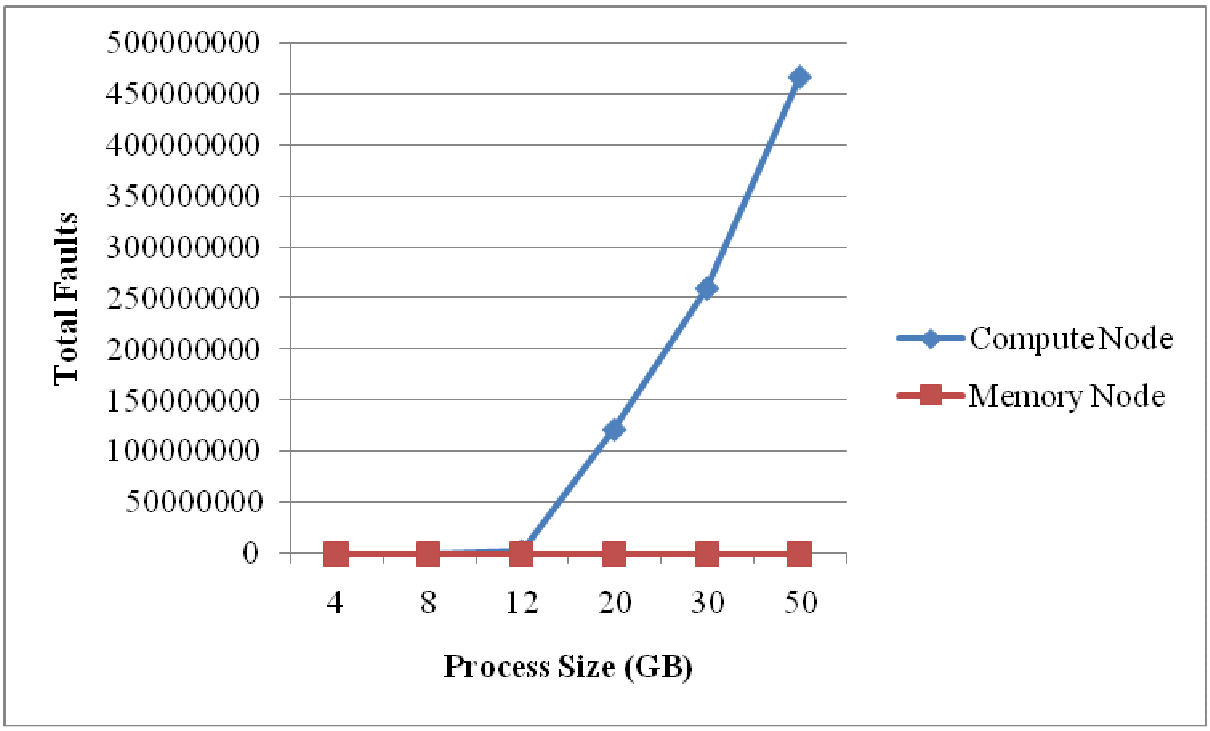

Figure 4.10 External: Process Size vs Total Faults

Out of all the faults that occurred during execution, a percentage of them will be page faults that require access to the disk while the others will be page reclaims that will not. Table 4.11 shows the averages for the number of page faults that occurred during execution, and Figure 4.11 illustrates the trends that these averages took as the process size increased. At the same time, Table 4.12 shows the averages of the page fault percentages for each set of trials where faults occurred, and Figure 4.12 illustrates the trend that they follow.

Table 4.11 External: Process Size vs Page Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	59893	580534	36026299	54724757	80756267
Memory Node	0	0	0	0	0	0

Table 4.12 External: Page Fault Percentage

Process Size (GB)	8	12	20	30	50
Percentage Page Faults	13.15	17.61	29.47	21.01	17.27

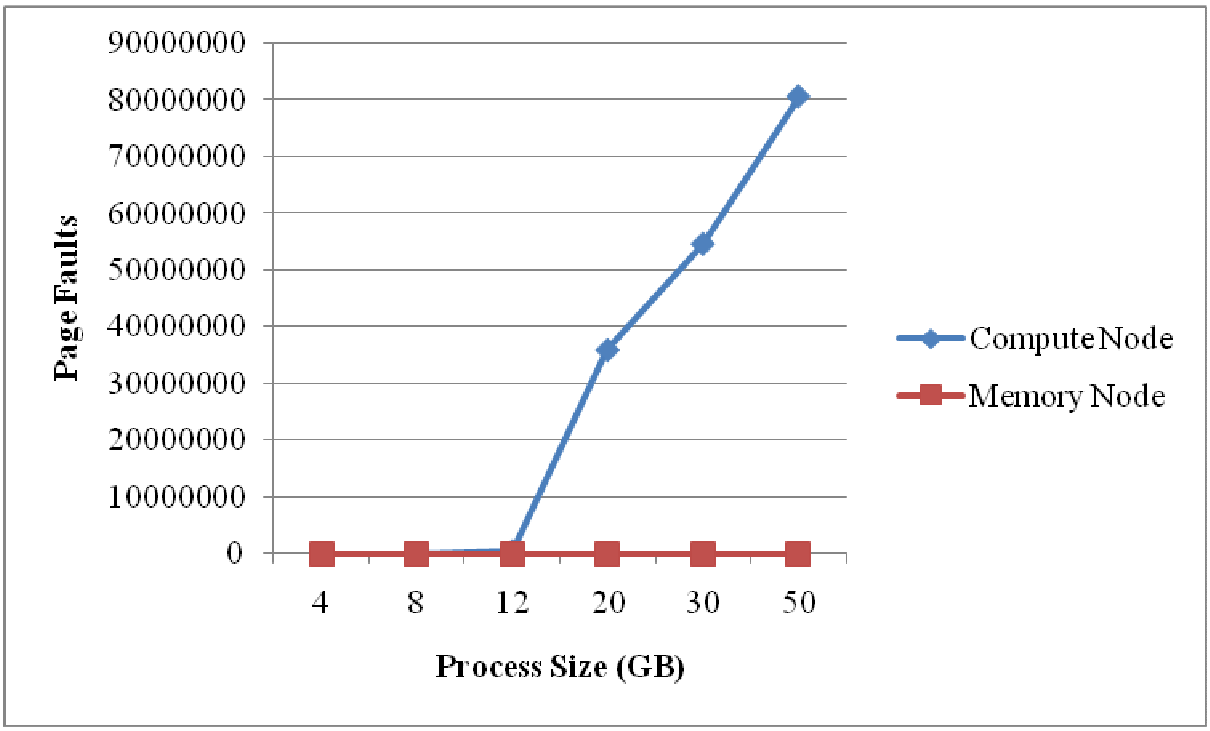

Figure 4.11 External: Process Size vs Page Faults

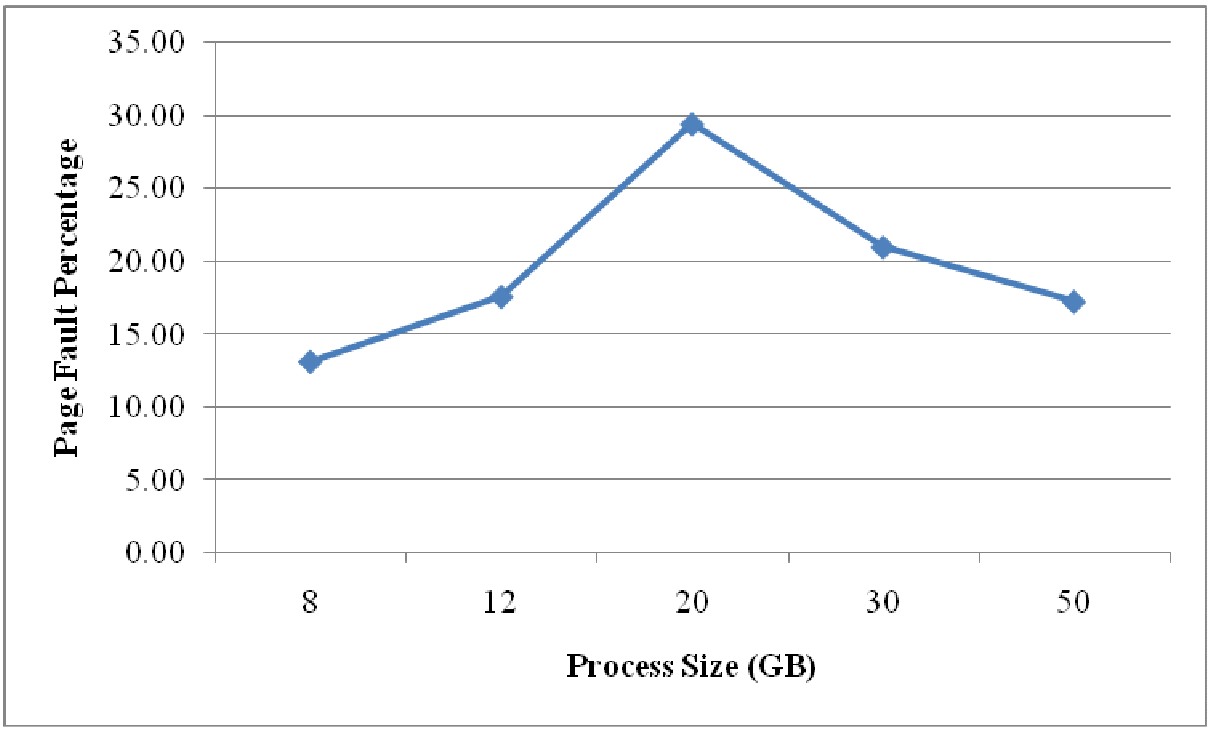

Figure 4.12 External: Page Fault Percentage

As the process size increases, the number of faults increases, and therefore, the number of page faults increases, as verified by these data. However, the page fault percentage did not stay constant throughout all process sizes. This percentage peaked around the 20 GB process size at 29 percent. Afterward, the percentage decreased as the process size decreased. One page fault can be serviced in the same amount of time as approximately one million page reclaims, so a change in the distribution of page faults within the total faults influences the total execution time. Therefore, even though the total number of faults increases as the process size increases, if the page fault percentage decreases, the execution time will not increase as rapidly.

4.3 Dynamic Allocation

Now the version of the program that used dynamic allocation will be examined. Dynamic allocation is different from static and external allocation since data allocated using malloc() is stored on the heap. However, the differences between dynamic allocation and static and external allocation occur while allocating the memory, not while executing the matrix multiplication function. Because of this, the results gathered from this set of trials did not differ significantly. Table 4.13 lists the averages for the total execution times for the matrix multiplication function, and Figure 4.13 shows the trends that the averages listed followed as the process size increased. As the process size increased, the execution time increased linearly throughout all trials on the memory node. Execution times on the compute node trials did not follow this trend. Instead, there was a large increase in execution time starting around the point when the process became larger than main memory. The memory node showed significant speedups on average when compared to the compute node.

Table 4.13 Dynamic: Process Size vs Execution Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	163	2000	177954	218110	248692
Memory Node Time (s)	49	98	148	247	370	619

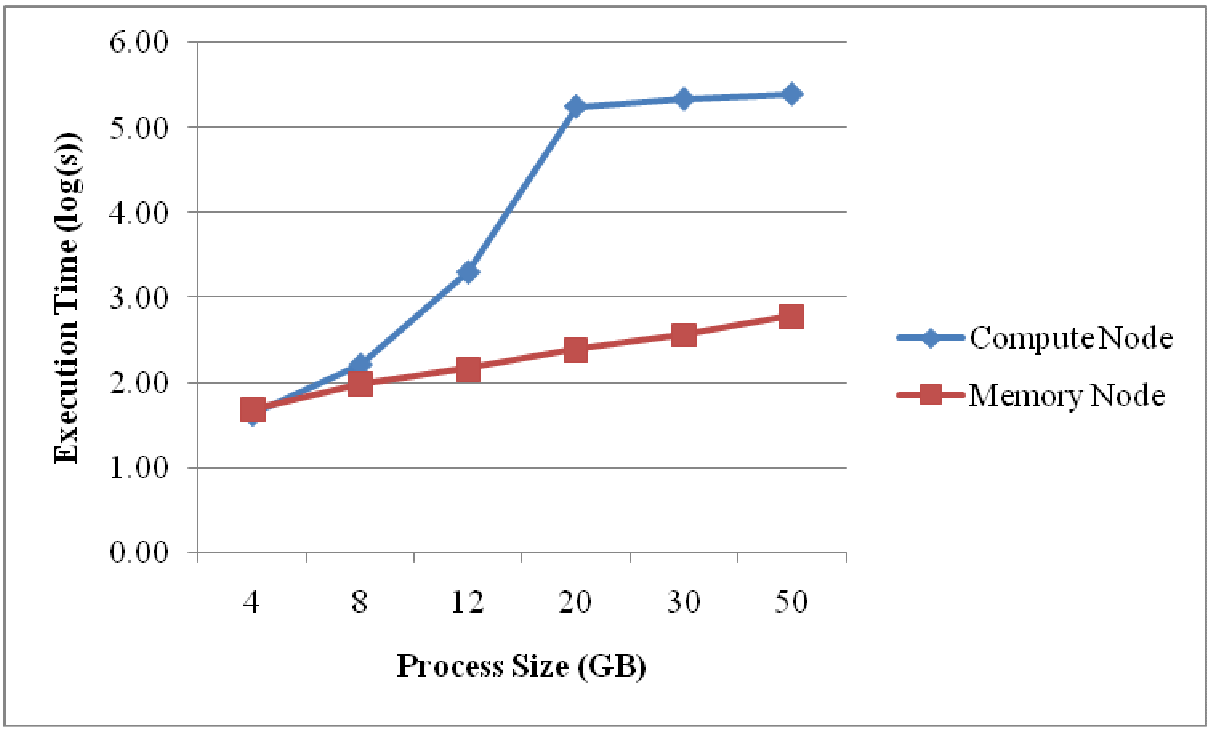

Figure 4.13 Dynamic: Process Size vs Execution Time

The dramatic increase in execution time for the compute node occurred when the total process size first exceeded the size of the compute node's main memory. This was expected because once the process size exceeds the size of main memory, the operating system must swap pages to the swap device, and it can take up to six orders of magnitude more time for the operating system to fetch data from the hard disk as it does to fetch data from main memory. After the process size reached 20 GB, the execution time still increased, but at a lower rate. There were no processes larger than the memory node's main memory, so there was no drastic increase in execution time. The largest measured speedup occurred when the process size was 20 GB, where a speed up of approximately 720. At a process size of 50 GB, a speedup of approximately 402 was measured. This showed a clear advantage of using a memory node for large processes rather than a compute node.

Table 4.14 lists the averages for the amount of time the matrix multiplication function spent in user mode, and Figure 4.14 shows the trends that these averages followed as process sizes increased. These results show that there was not a large difference in the amount of time spent in user mode between the two types of nodes. These results indicated that the two types of nodes performed computations at approximately the same speed.

Table 4.14 Dynamic: Process Size vs User Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	87	137	232	311	488
Memory Node Time (s)	49	98	148	247	370	619

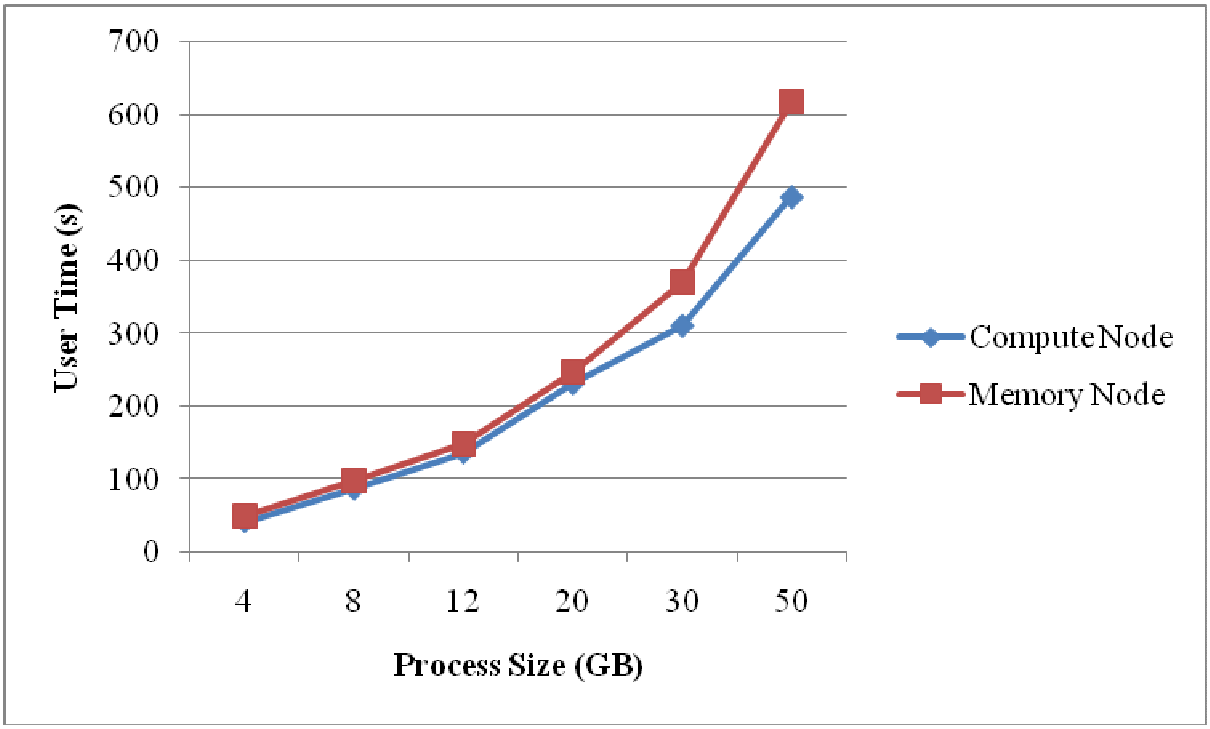

Figure 4.14 Dynamic: Process Size vs User Time

Table 4.15 lists the averages for the time that the matrix multiplication function spent in kernel mode, and Figure 4.15 illustrates the trends that these averages followed as process sizes increased. There were processes being executed that were larger than main memory for the compute node, so for those processes, the entire process cannot be stored within main memory, and the process would need to go into kernel mode to service memory-related interrupts. As the process size increased past the size of the compute node's main memory, the time spent in kernel mode increased as well. Since the memory node's main memory was larger than any process size used during this study, there was no need for its processes to enter kernel mode to page in data, and the results show that they did not.

Table 4.15 Dynamic: Process Size vs System Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	0	7	65	2226	3653	5877
Memory Node Time (s)	0	0	0	0	0	0

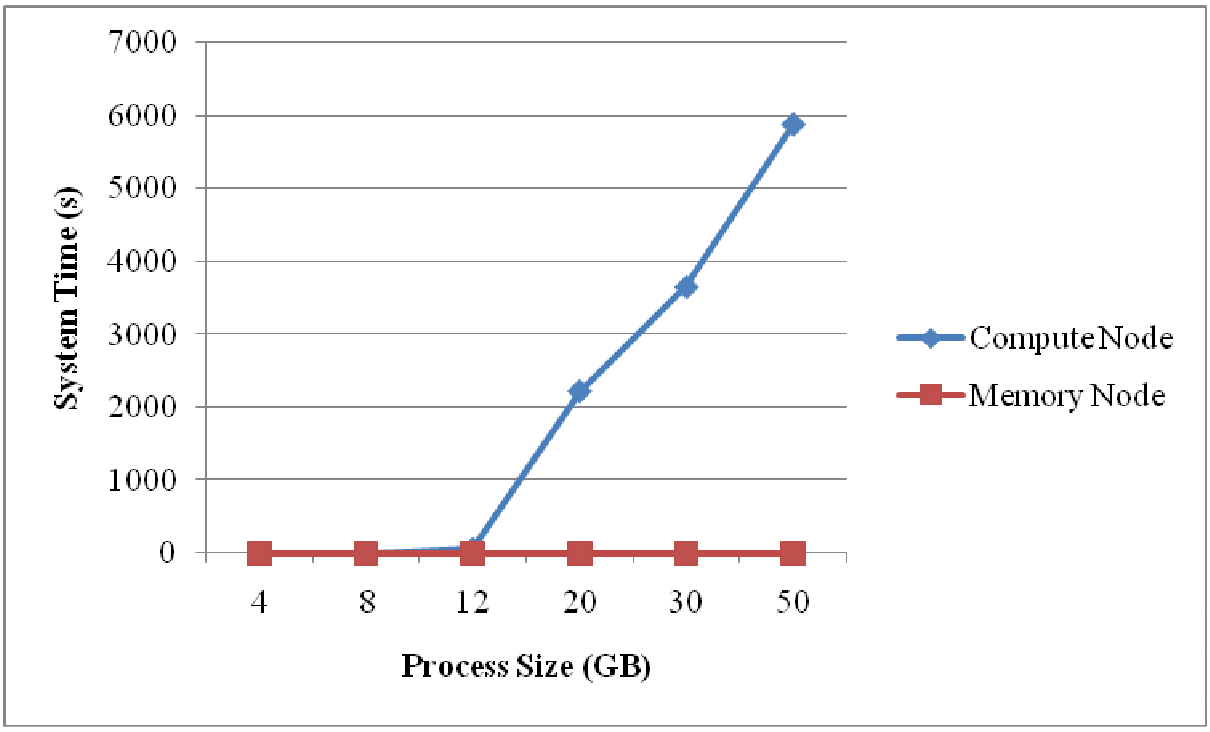

Figure 4.15 Dynamic: Process Size vs System Time

Another aspect important in comparing the two types of nodes is the number of faults that occurred during execution. Table 4.16 shows the average numbers of faults that occurred during execution, and Figure 4.16 shows the trends that these averages followed with increasing process sizes. On the compute node trials, as the process size increased after surpassing the size of the compute node's main memory, the number of faults that occurred during those trials increased, as expected. Also, no faults occurred during the execution of the matrix multiplication function on trials executed on the memory node, which was expected as well. These results reiterated that if the process size is larger than the size of main memory, then the entire process cannot fit within main memory, and when data cannot be found because of this, faults will occur.

Table 4.16 Dynamic: Process Size vs Total Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	449128	3110130	122186077	256637339	467985120
Memory Node	0	0	0	0	0	0

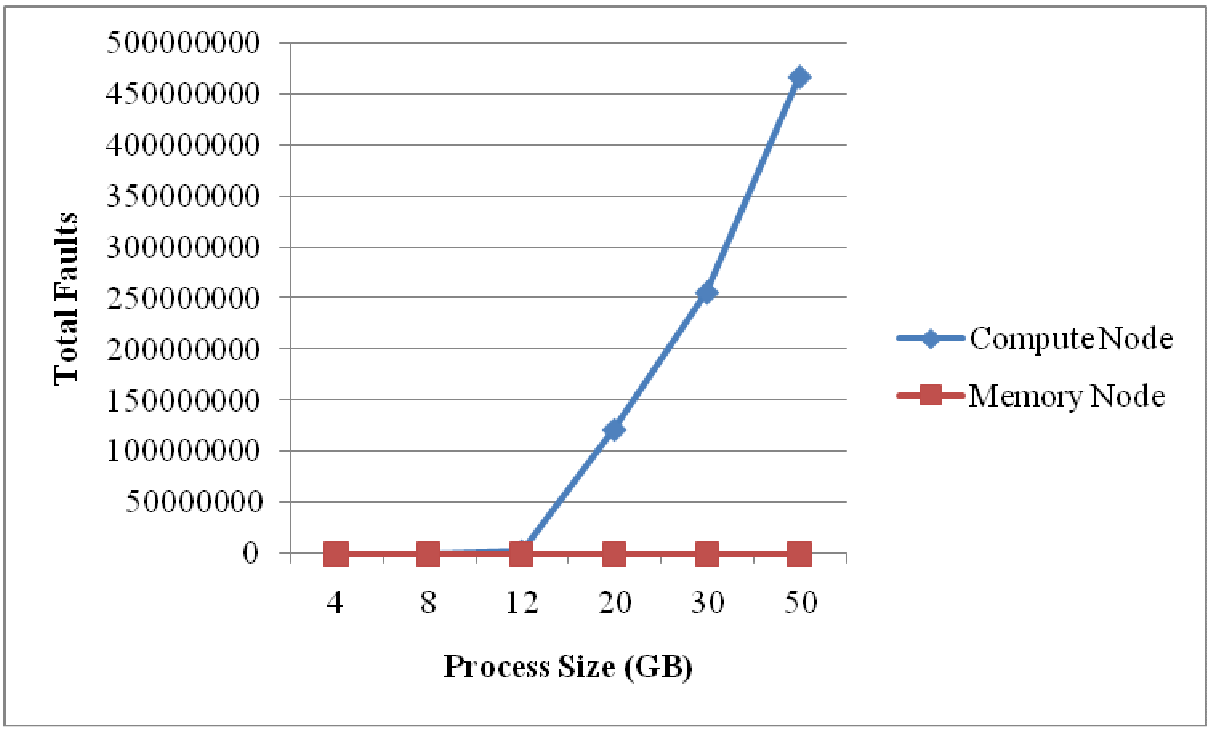

Figure 4.16 Dynamic: Process Size vs Total Faults

Out of all the faults that occurred during execution, a percentage of them will be page faults and require access to the disk while the others will not. Table 4.17 shows the averages for the number of page faults that occurred during execution, and Figure 4.17 illustrates the trends that these averages took as the process size increased. At the same time, Table 4.18 shows the averages of the page fault percentages for each set of trials where faults occurred, and Figure 4.18 illustrates the trend that they follow.

Table 4.17 Dynamic: Process Size vs Page Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	58894	527622	35931348	54541881	80108558
Memory Node	0	0	0	0	0	0

Table 4.18 Dynamic: Percentage Page Faults

Process Size (GB)	8	12	20	30	50
Percentage Page Faults	13.11	16.86	29.41	21.25	17.12

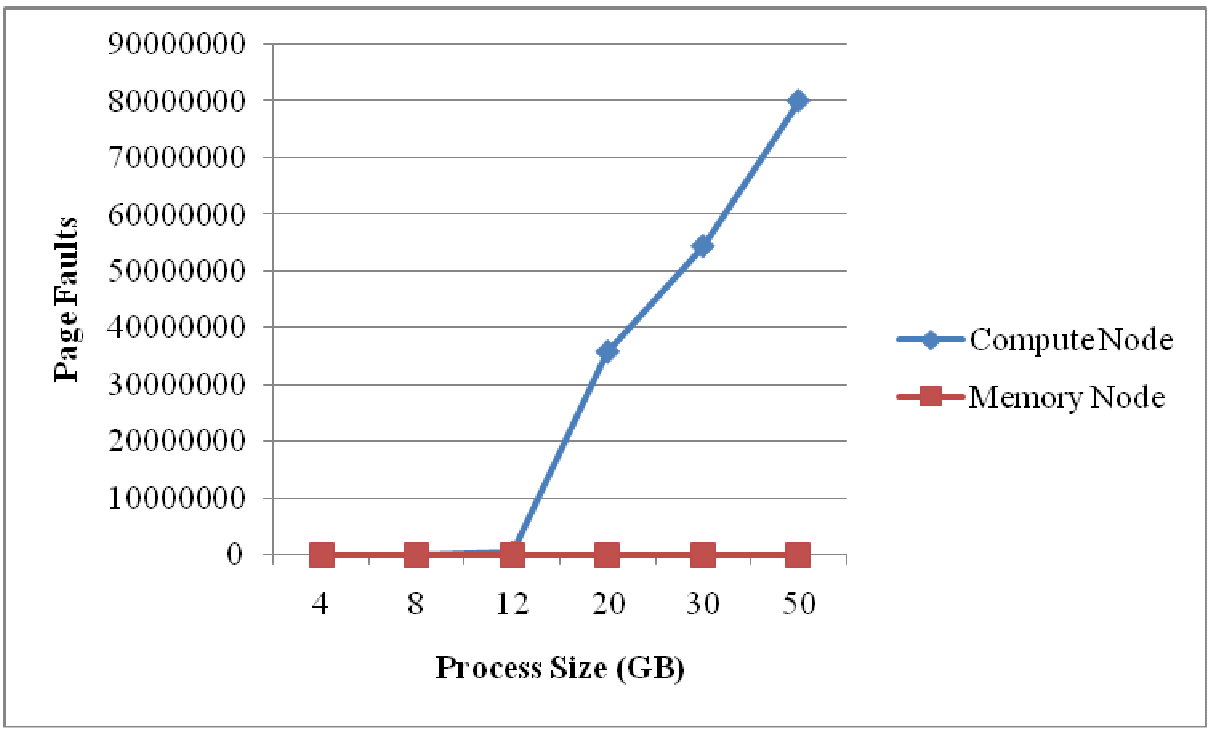

Figure 4.17 Dynamic: Process Size vs Page Faults

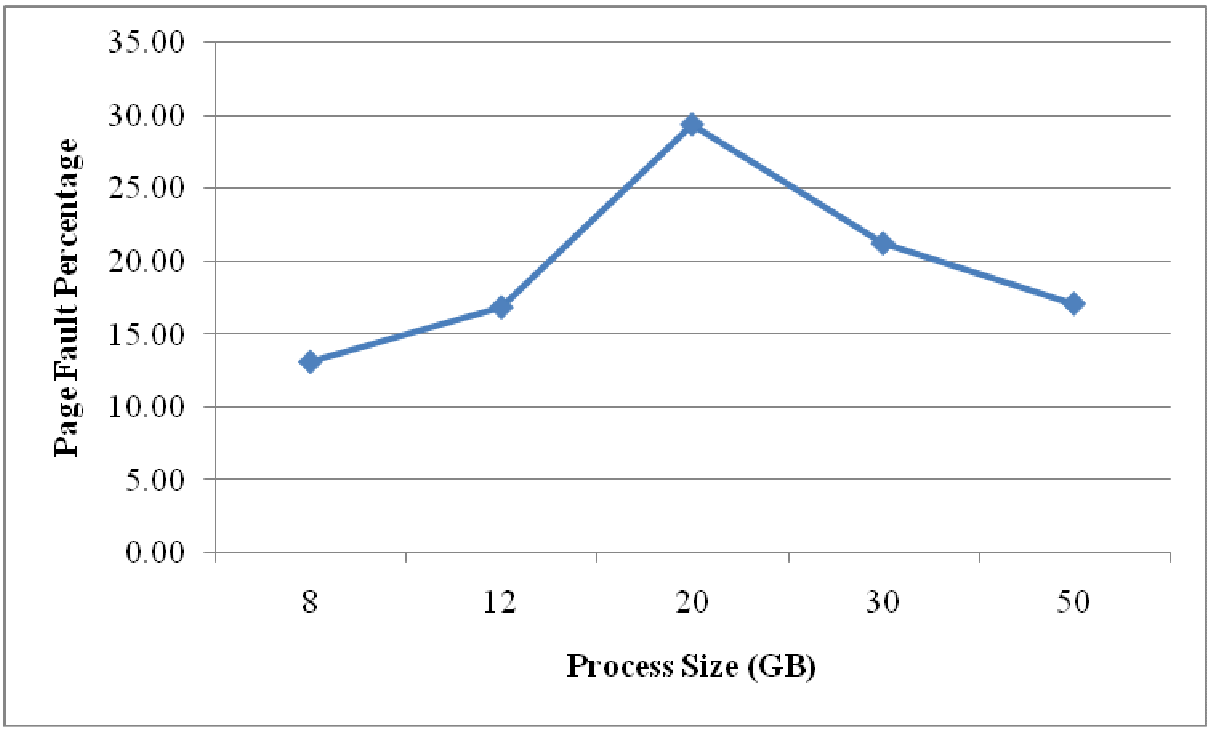

Figure 4.18 Dynamic: Page Fault Percentage

As the process size increases, the number of faults increases, and therefore, the number of page faults increases, as verified by these data. However, the percentage of total faults that occurred that were page faults did not stay constant. The percentage peaked around the 20 GB process size at 29 percent. This distribution was similar to the percentage distributions for the static and external allocation programs, and as with those setups, the percentage is significant because page faults take much longer to service than page reclaims, so changes in the distribution would affect the growth of the execution time as process sizes increase. Therefore, even though the total number of faults increases as the process size increases, if the page fault percentage decreases, the execution time will not increase as rapidly.

4.4 Automatic Allocation

Lastly, the version of the program that used automatic allocation will be examined. Automatic allocation is different from the previously mentioned types of allocation because unlike the others, automatic allocation stores variables in the stack region of the C program process model. Any impact on the results caused by this difference, however, was unable to cause any significant differences between results gathered from this set of trials and results from the sets of trials for the other three types of memory allocation. Table 4.19 lists the averages for the total execution times for the matrix multiplication function, and Figure 4.19 shows the trends that the averages listed followed as the process size increased. As the process size increased, the execution time increased linearly throughout all trials. Execution times on the compute node trials did not follow this trend. Instead, there was a large increase in execution time starting around the point when the process became larger than main memory. The memory node showed significant speedups on average when compared to the compute node.

Table 4.19 Automatic: Process Size vs Execution Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	166.1	1796.2	172990	217222	265334
Memory Node Time (s)	49	98	147	247	370	623

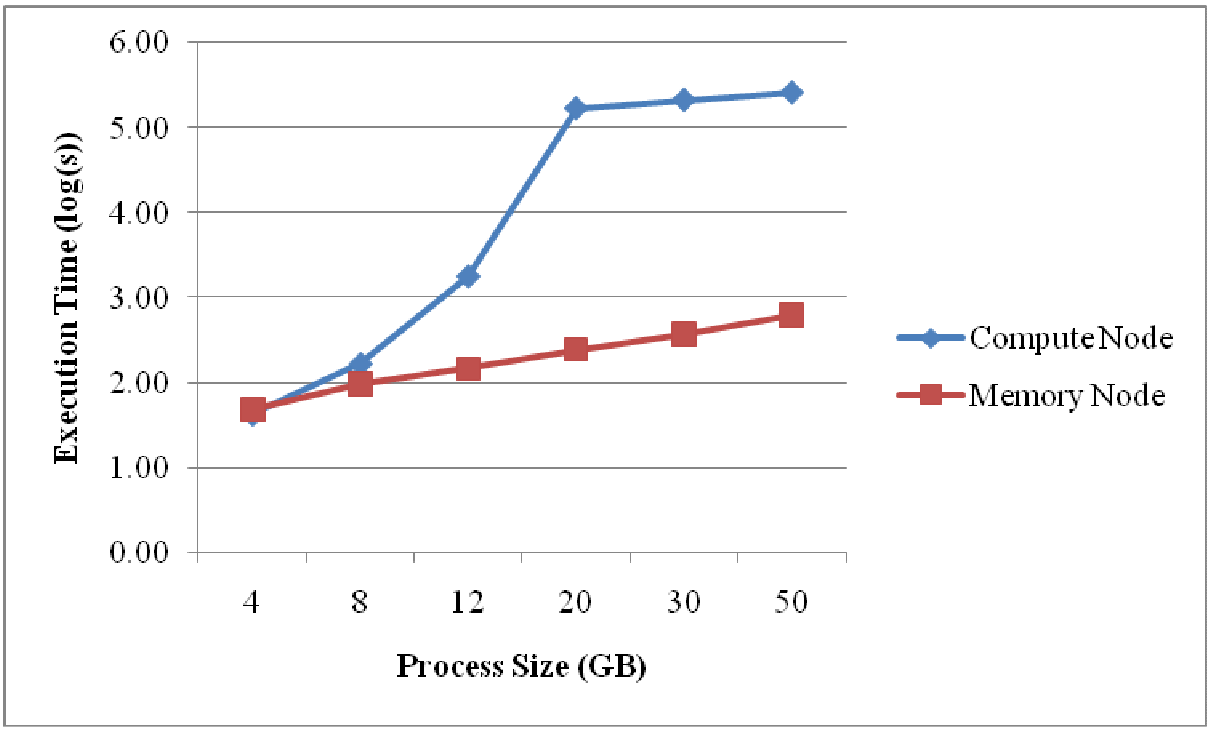

Figure 4.19 Automatic: Process Size vs Execution Time

The dramatic increase in execution time for the compute node occurred when the total process size first exceeded the size of the compute node's main memory. If the process size surpasses the size of main memory, the entire process cannot be kept within main memory, so a swap device must be used. It can take up to six orders of magnitude longer to access the hard disk than main memory, so a dramatic increase in execution time was expected. There were no processes larger than the memory node's main memory, so there was no drastic increase in execution time. The largest measured speedup occurred when the process size was 20 GB, where a speedup of approximately 700. At a process size of 50 GB, a speedup of approximately 425 was measured. This showed a clear advantage of using a memory node for large processes rather than a compute node.

Table 4.20 shows the averages for the amount of time the matrix multiplication function spent in user mode, and Figure 4.20 illustrates the trends that these averages followed as process sizes increased. These results show that there was not a large difference in the amount of time spent in user mode between the two types of nodes. These results indicated that the two types of nodes performed computations at approximately the same speed.

Table 4.20 Automatic: Process Size vs User Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	42	87	136	232	312	490
Memory Node Time (s)	49	98	147	246	370	623

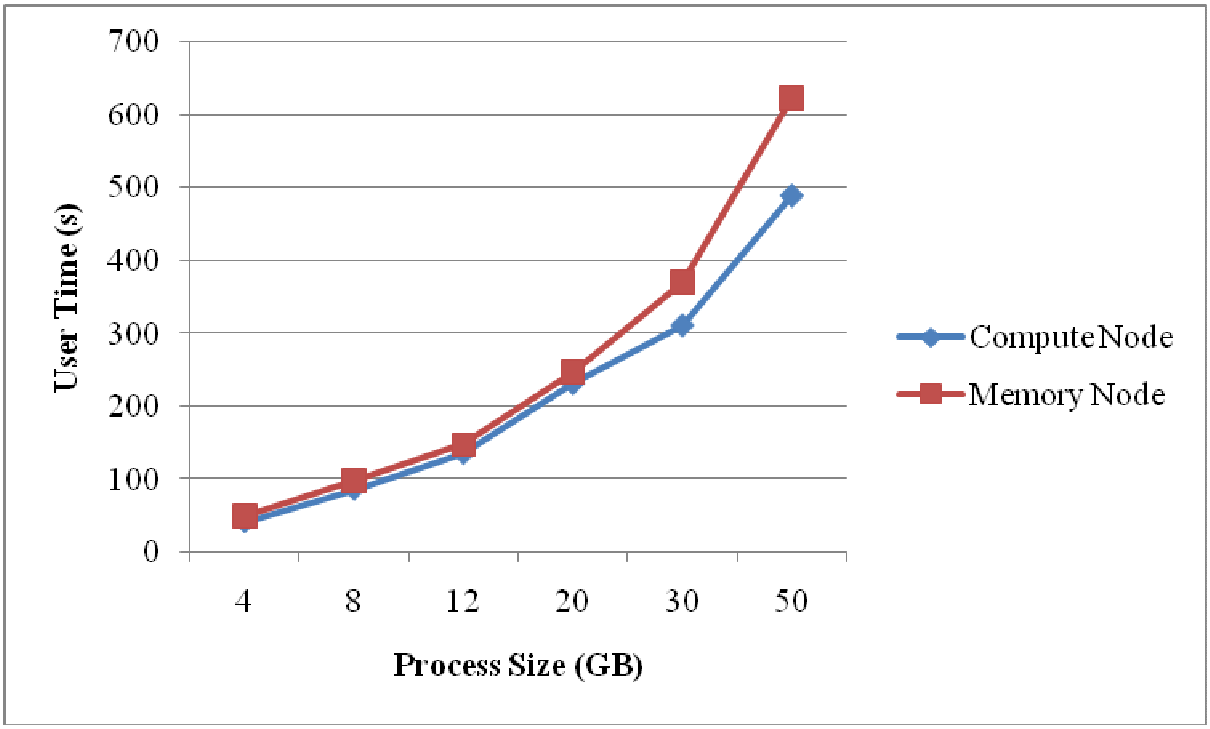

Figure 4.20 Automatic: Process Size vs User Time

Table 4.21 lists the averages for the time that the matrix multiplication function spent in kernel mode, and Figure 4.21 illustrates the trends that these averages followed as process sizes increased. There were processes being executed that were larger than main memory for the compute node, so for those processes, the entire process cannot be stored within main memory, and the process would need to go into kernel mode to service memory-related interrupts. As the process size increased past the size of the compute node's main memory, the time spent in kernel mode increased as well. Since the memory node's main memory was larger than any process size used during this study, there was no need for its processes to enter kernel mode, and the results show that they did not.

Table 4.21 Automatic: Process Size vs System Time

Process Size (GB)	4	8	12	20	30	50
Compute Node Time (s)	0	7	61	2210	3660	5944
Memory Node Time (s)	0	0	0	0	0	0

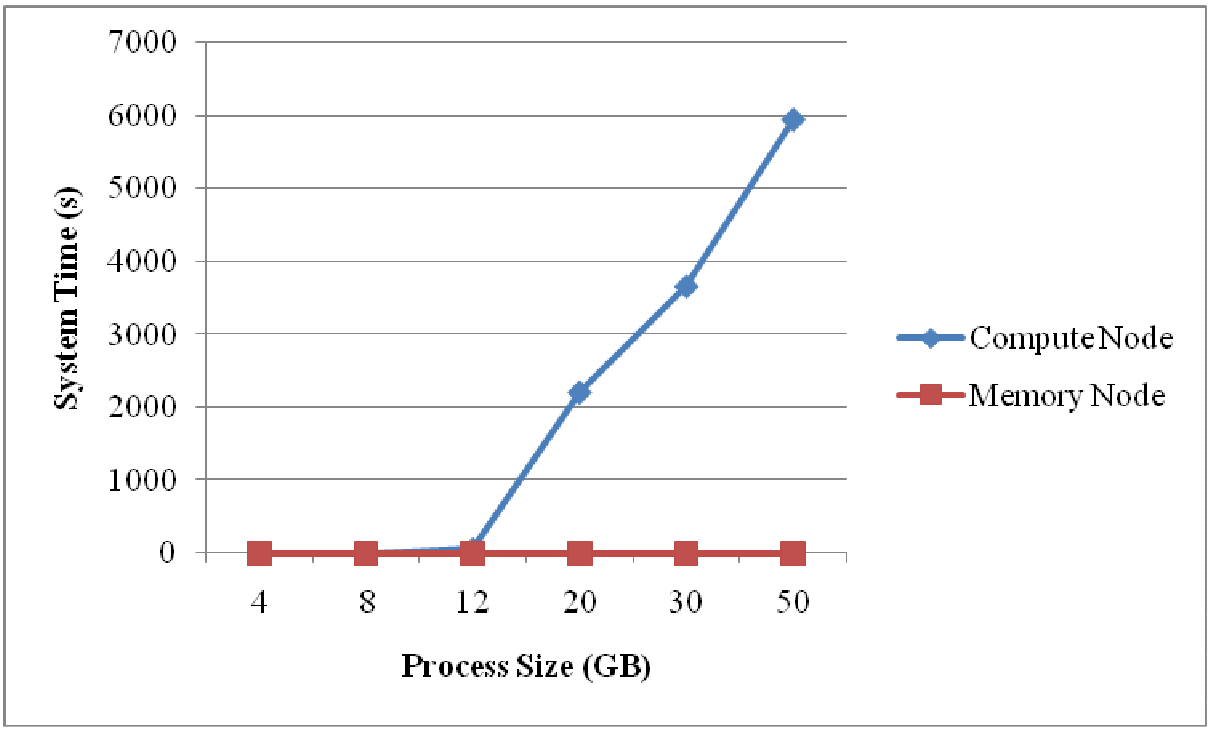

Figure 4.21 Automatic: Process Size vs System Time

Another aspect important in comparing the two types of nodes is the number of faults that occurred during execution. Table 4.22 shows the average numbers of faults that occurred during execution, and Figure 4.22 shows the trends that these averages followed with increasing process sizes. As expected, the number of page faults increased on compute node trials as the process size increased past the size of main memory. No faults occurred during the execution of the matrix multiplication function on trials executed on the memory node, which was expected as well. These results reiterated that if the process size is larger than the size of main memory, then the entire process cannot fit within main memory, and when data cannot be found in main memory because of this, faults will occur.

Table 4.22 Automatic: Process Size vs Total Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	451731	3009404	121797743	256867998	467417735
Memory Node	0	0	0	0	0	0

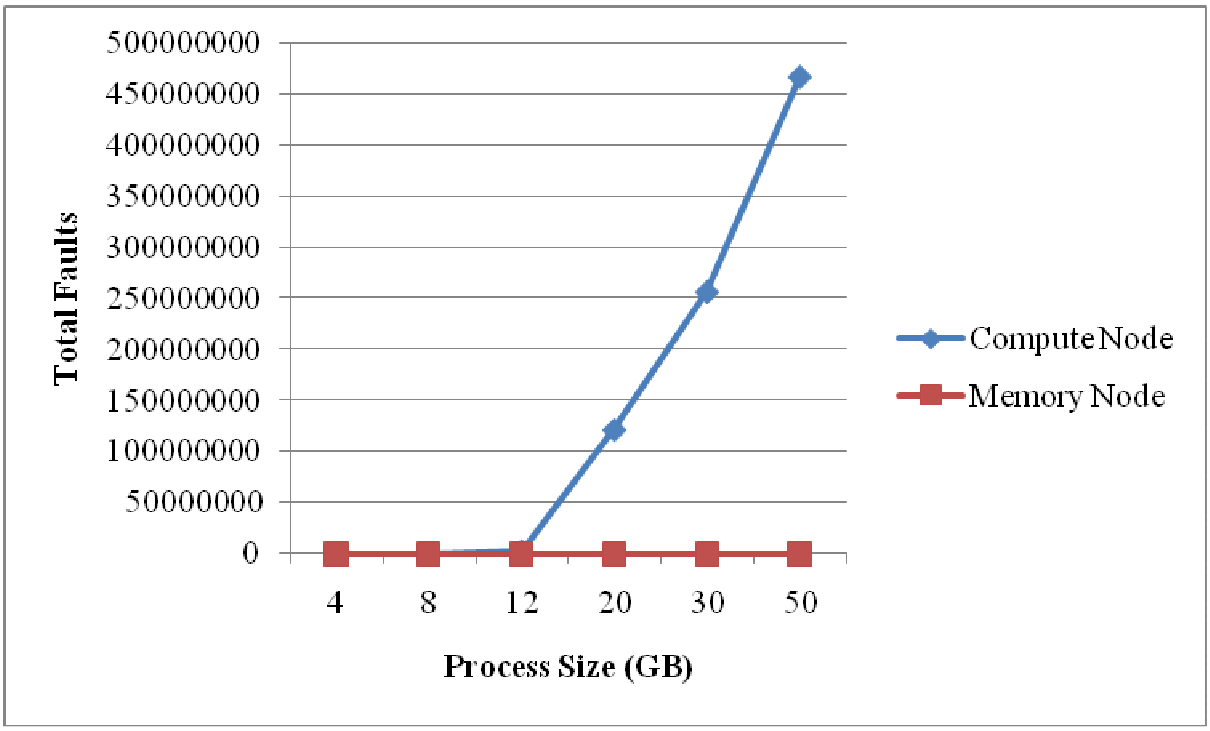

Figure 4.22 Automatic: Process Size vs Total Faults

Out of all the faults that occurred during execution, a percentage of them will be page faults and require access to the disk while the others will not. Table 4.23 shows the averages for the number of page faults that occurred during execution, and Figure 4.23 illustrates the trends that these averages took as the process size increased. At the same time, Table 4.24 shows the averages of the page fault percentages for each set of trials where faults occurred, and Figure 4.24 illustrates the trend that they follow.

Table 4.23 Automatic: Process Size vs Page Faults

Process Size (GB)	4	8	12	20	30	50
Compute Node	0	59712	492932	35481760	54446747	81808949
Memory Node	0	0	0	0	0	0

Table 4.24 Automatic: Page Fault Percentage

Process Size (GB)	8	12	20	30	50
Percentage Page Faults	13.22	16.26	29.13	21.20	17.50

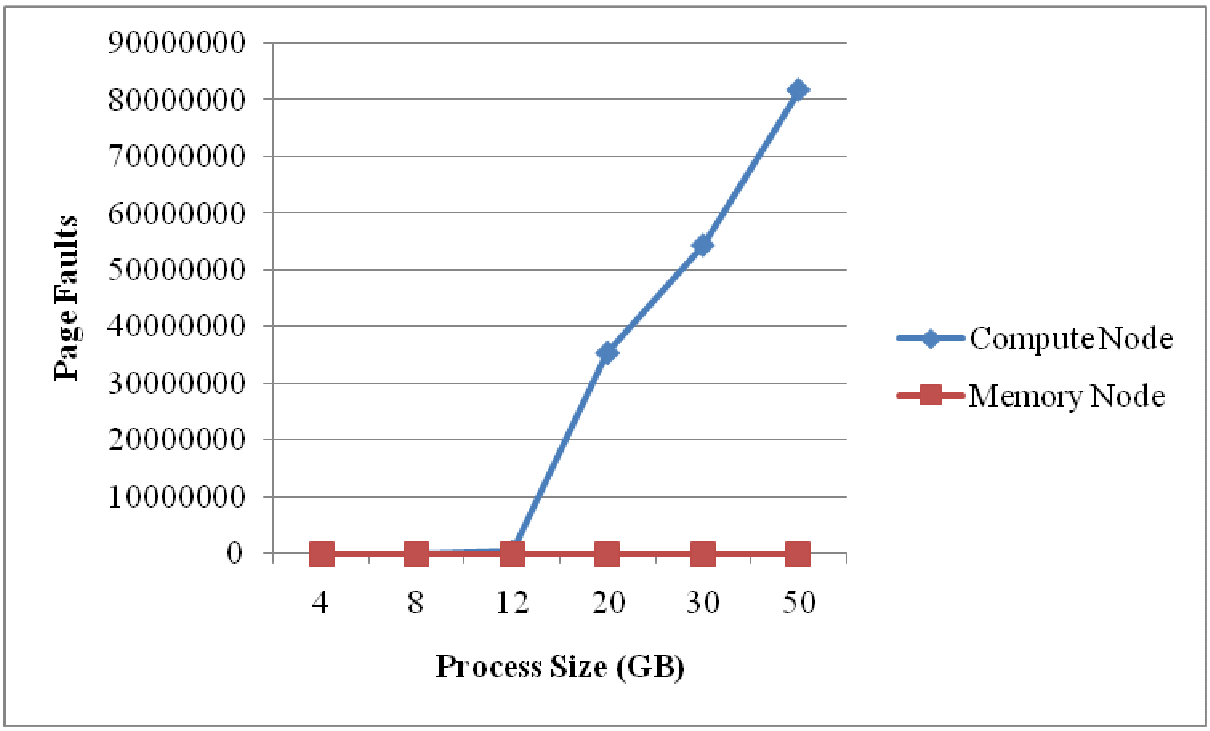

Figure 4.23 Automatic: Process Size vs Page Faults

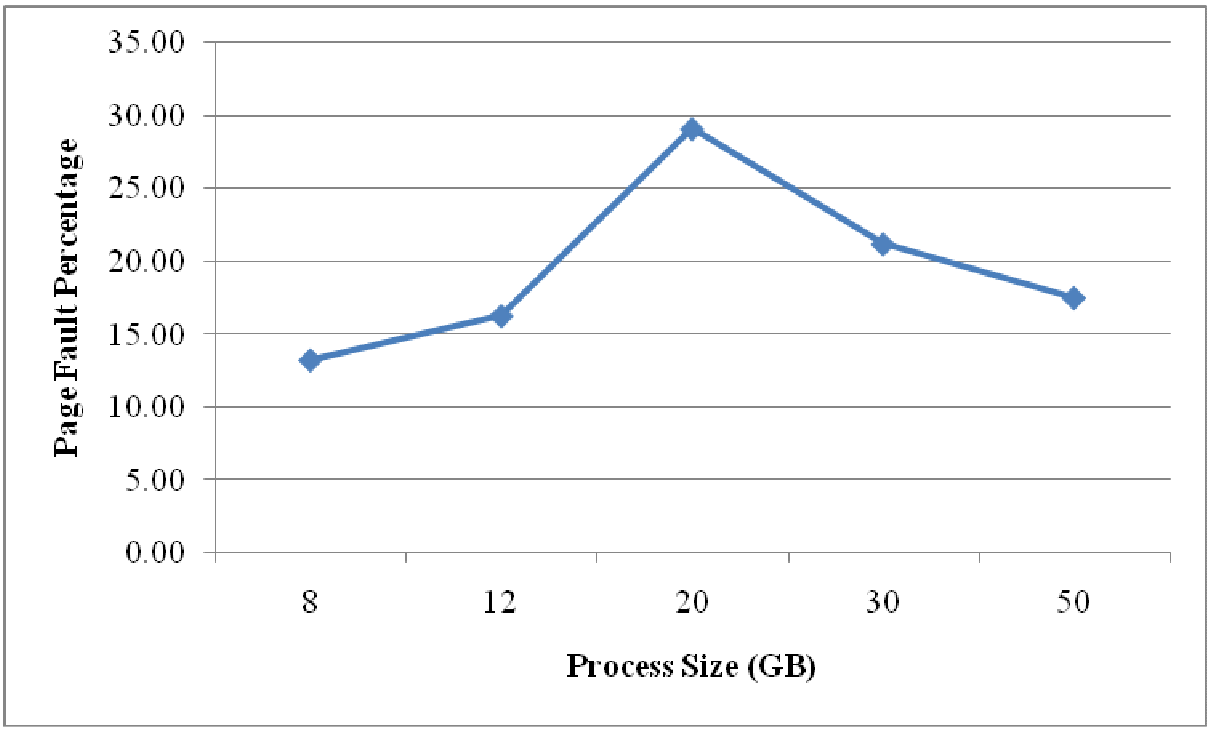

Figure 4.24 Automatic: Page Fault Percentage

As the process size increases, the number of faults increases, and therefore, the number of page faults increases, as verified by these data. However, the page fault percentage did not stay constant. The percentage peaked around the 20 GB process size at 29 percent. This distribution was similar to the percentage distributions for the static and external allocation programs, and as with those setups, the percentage is significant because page faults take much longer to service than page reclaims, so changes in the distribution would affect the growth of the execution time as process sizes increase. Therefore, even though the total number of faults increases as the process size increases, if the page fault percentage decreases, the execution time will not increase as rapidly.

4.5 Conclusions

It is possible to compare the two types of cluster nodes using the observations made during this experiment. The conclusions drawn from this study can be separated into three groups: timing, swapping, and programming.

For all methods of memory allocation used in this experiment, several observations were made regarding the trends for the timing of this experiment. First, when processes are small and can fit in main memory with room to spare, the amount of main memory the machine contains did not seem to have an effect. On the other hand, when processes were larger than main memory, the execution time increased dramatically as the process had to swap data back and forth to the swap device. In addition, computations were performed at the same speed regardless of the size of main memory. When comparing the execution times of the compute node trials and the memory node trials, the compute node times were much larger when processes were larger. Speedups with the memory node were not constant, but peak speedups of approximately 700 were recorded and even though the speedup decreased after a certain point, large speedups of around 400 were still observed at the largest process size. Based on these findings, regarding timing, it can be concluded that for small processes, there is no need to use a memory node. The benefit to using a memory node comes when processes larger than the compute node's main memory are being executed. With the speedups observed during his experimentation, a memory node can make a significant difference in research by executing, in a few minutes, a program that can potentially take three days on a compute node, allowing for more runs in less time.

In addition to providing timing results, this study also showed what would happen if a process is forced to swap memory during execution. Efficiency is a factor that is taken into consideration when running programs on a computer. Ideally, a CPU would fetch data and

instructions at zero latency and perform computations. Unfortunately, this is not the case. While it takes time to perform computations, time must also be spent just gathering data from memory. During this experimentation, the time spent to execute the matrix multiplication function was nearly the same as the time spent in User Mode and Kernel Mode combined. This indicates that nearly all time spent running a program on the memory node goes towards computations. Subtracting the User Mode and Kernel Mode times from the total execution time for trials on the compute node, however, indicate that the majority of time spent running the program is not spent on calculations, but rather on fetching data, which in itself, accomplishes nothing. For a process size of 20 GB, around 1.4 percent of the execution time was spent actually doing calculations. Therefore, it can be concluded that another benefit of using memory nodes is the greater efficiency gained by using them rather than using compute nodes.

The final conclusion relates to the different types of memory allocation used during this experimentation. All four methods of memory allocation: static, external, dynamic, and automatic yielded very similar results with no significant differences. Therefore, the conclusion is that the method of memory allocation is not influential on the execution times for large processes resembling the test programs, program that do not need to be distributed and loop large amounts of similar instructions. However, extra steps had to be taken to execute the version of the program using automatic allocation, namely, altering security files for the operating system. If the method of allocation is not influential, then it would be beneficial to the programmer to use static, external, or dynamic allocation when programming to avoid the unnecessary hardship that comes with using automatic allocation.

4.6 Future Work

Future work can include sets of trials that use different types of test programs than the ones used for this study such as programs that require distribution or programs that do not loop as heavily as this one did. By performing these tests, it would be possible to see if there would be any other effects on execution times and fault data, or if these programs yield similar results. Additionally, to further test the efficiency of using a memory node against the efficiency of using a compute node, power usage measurements can be taken while executing this study's experimental setup. This way, a figure of the number of calculation per watt can be obtained, yielding another measure of efficiency.

List of References

[Abd05] Abd-El-Barr, Mostafa, and Hesham El-Rewini. Fundamentals of Computer Organization and Architecture. New Jersey: John Wiley & Sons, Inc, 2005.

[And95] Anderson, Thomas E, David E. Culler, and David A. Patterson. "A Case for NOW (Network of Workstations)." IEEE Micro. Vol 15(1), February 1996: pp. 54-64.

[Bac90] Bach, Maurice J. The Design of the Unix Operating System. New Jersey: Prentice Hall, 1990.

[Ift93] Iftode, Liviu, Kai Li, and Karin Petersen. "Memory Servers for Multicomputers." Proceedings of the 38th IEEE International Conference. 1993: pp. 534-547.

[Lit88] Litzkow, Michael J, Miron Livny, and Matt W. Mutka. "Condor – A Hunter of Idle Workstations." Proceedings of the 38th IEEE International Conference. June 1988: pp. 104-111.

[Rom03] Romero, Rodrigo, and David H. Williams. "Performance Requirements of Dedicated Distributed Memory Servers." Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. 2003: pp 1192-1197.

[Sch00] Schildt, Herbert. C: The Complete Reference, Fourth Edition. California: Osborne/McGraw-Hill, 2000.

[Som08] Somanathan, Muthuveer. "Towards an Adaptation Heuristic for the Linux 2.6 Virtual Memory Manager (VMM) Parameter Swap Cluster Max (SCM) With Respect to program Behavior: A First Step." Master's Thesis. El Paso, Texas: The University of Texas, El Paso, 2008.

[Sta05] Stallman, Richard. Using the GNU Compiler Collection. Boston, Massachusetts: GNU Press, 2005.

[Tan08] Tanenbaum, Andrew S. Modern Operating Systems. Third Edition. New Jersey: Pearson Prentice Hall, 2008.

[Wil94] Williams, David H. Three Dialects of C: C, ANSI C, C++. 1994.

Appendix A

Full Data Set

The following is the complete set of data taken from this experimentation. As was stated in Chapter 4, the data in Appendix A will be organized in the same manner as in Chapter 4. This is intended for those interested in viewing the results for each individual trial in this study.

A.1 Static Allocation

Table A.1a Static, Compute Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	42	175	2369	174212	217652	256960
2	42	169	2504	173336	215882	257701
3	42	168	1137	173607		
4	42	169	2075	174445		
5	42	171	1238	174162		
6	42	166	1739	172236		
7	42	170	2204			
8	42	169	1901			
9	42	168	3270			
10	42	166	1226			
Average	42	169	1966	173666	216767	257331

Table A.1b Static, Memory Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	49	98	148	248	373	620
2	49	98	147	246	371	619
3	49	98	147	247	369	620
4	49	98	147	246	371	619
5	49	98	147	246	373	620
6	49	99	147	246	371	620
7	49	98	147	246	369	623
8	49	98	148	246	370	620
9	49	98	147	246	369	620
10	49	98	147	246	369	619
Average	49	98	147	246	371	620

Table A.2a Static, Compute Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	42	87	138	232	313	508
2	42	88	137	232	312	494
3	42	87	136	232		
4	42	87	137	231		
5	42	87	136	232		
6	42	87	136	232		
7	42	87	139			
8	42	87	137			
9	42	88	137			
10	42	87	136			
Average	42	87	137	232	313	501

Table A.2b Static, Memory Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	49	98	148	248	373	620
2	49	98	147	246	370	619
3	49	98	147	247	369	620
4	49	98	147	246	371	619
5	49	98	147	246	373	620
6	49	99	147	246	371	619
7	49	98	147	246	368	623
8	49	98	148	245	370	620
9	49	98	147	246	369	619
10	49	98	147	246	369	619
Average	49	98	147	246	370	620

Table A.3a Static, Compute Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	8	73	2225	3665	6270
2	0	7	74	2225	3662	5916
3	0	7	49	2222		
4	0	7	69	2218		
5	0	7	51	2226		
6	0	7	61	2217		
7	0	7	68			
8	0	7	63			
9	0	7	86			
10	0	7	51			
Average	0	7	65	2222	3664	6093

Table A.3b Static, Memory Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.4a Static, Compute Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	63047	581934	35543946	54411932	80235842
2	0	61626	607288	35586998	54350118	81110503
3	0	58649	388146	35604220		
4	0	60051	539166	35587801		
5	0	59636	406403	35665284		
6	0	58553	489387	35459421		
7	0	59716	553328			
8	0	59357	510088			
9	0	59081	717109			
10	0	59680	406542			
Average	0	59940	519939	35574612	54381025	80673173

Table A.4b Static, Memory Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.5a Static, Compute Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	406842	2687719	86153922	202317886	387488660
2	0	401835	2720843	86929441	202553569	386550385
3	0	383742	2200717	86419914		
4	0	388908	2587669	86080029		
5	0	387959	2268214	86846687		
6	0	384657	2534460	86542431		
7	0	387914	2784270			
8	0	389361	2613208			
9	0	388491	2970815			
10	0	390044	2272822			
Average	0	390975	2564074	86495404	202435728	387019523

Table A.5b Static, Memory Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

A.2 External Allocation

Table A.6a External, Compute Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	42	172	3633	179142	224898	255372
2	42	165	2281	177471	213709	
3	42	164	3866			
4	42	166	1306			
5	42	166	1302			
6	42	167	3473			
7	42	169	3076			
8	43	167	1220			
9	42	163	1972			
10	42	167	1774			
Average	42	167	2390	178307	219304	255372

Table A.6b External, Memory Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	49	99	148	248	373	626
2	49	99	148	247	373	626
3	49	99	148	248	373	626
4	49	99	148	248	373	626
5	49	99	149	248	375	626
6	49	99	148	249	373	626
7	49	99	148	248	373	627
8	49	99	148	248	374	627
9	49	99	149	248	374	626
10	49	99	148	248	373	629
Average	49	99	148	248	373	627

Table A.7a External, Compute Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	42	87	138	232	314	494
2	42	87	136	232	312	
3	42	87	140			
4	42	87	135			
5	42	87	136			
6	42	87	140			
7	42	88	138			
8	43	88	135			
9	42	87	137			
10	42	87	136			
Average	42	87	137	232	313	494

Table A.7b External, Memory Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	49	99	148	248	373	626
2	49	99	148	247	373	626
3	49	99	148	248	373	626
4	49	99	148	248	373	626
5	49	99	149	248	375	626
6	49	99	148	249	373	626
7	49	99	148	248	373	627
8	49	99	148	248	374	627
9	49	99	149	248	374	626
10	49	99	148	248	373	629
Average	49	99	148	248	373	627

Table A.8a External, Compute Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	7	89	2231	3676	5909
2	0	7	69	2224	3666	
3	0	7	92			
4	0	7	54			
5	0	7	53			
6	0	7	85			
7	0	7	81			
8	0	7	51			
9	0	7	65			
10	0	7	62			
Average	0	7	70	2228	3671	5909

Table A.8b External, Memory Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.9a External, Compute Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	60367	762409	36090246	55266899	80756267
2	0	59196	568317	35962352	54182615	
3	0	57540	796527			
4	0	59109	419954			
5	0	60268	413276			
6	0	60155	746322			
7	0	62795	685514			
8	0	60997	404704			
9	0	58115	512941			
10	0	60387	495374			
Average	0	59893	580534	36026299	54724757	80756267

Table A.9b External, Memory Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.10a External, Compute Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	393698	3041745	86112226	200858220	386921058
2	0	390667	2734468	86357654	210920251	
3	0	382654	3050152			
4	0	391585	2237389			
5	0	399331	2311388			
6	0	398427	3086148			
7	0	414366	2871630			
8	0	403141	2268809			
9	0	384549	2496204			
10	0	398262	2551209			
Average	0	395668	2664914	86234940	205889236	386921058

Table A.10b External, Memory Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

A.3 Dynamic Allocation

Table A.11a Dynamic, Compute Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	42	165	1643	179414	215959	246686
2	42	163	1220	179057	220261	251899
3	42	161	2610	179607		247491
4	42	164	2592	174730		
5	42	163	1115	175565		
6	42	166	1816	179353		
7	42	163	2511			
8	42	161	1923			
9	42	162	2378			
10	42	162	2189			
Average	42	163	2000	177954	218110	248692

Table A.11b Dynamic, Memory Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	49	98	148	246	371	616
2	49	98	148	246	370	620
3	49	98	148	247	370	617
4	49	98	148	247	370	622
5	49	98	148	247	371	624
6	49	98	148	247	370	617
7	49	98	148	247	370	616
8	49	98	148	247	371	624
9	49	98	148	247	370	616
10	49	98	148	246	370	617
Average	49	98	148	247	370	619

Table A.12a Dynamic, Compute Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	42	87	137	232	310	487
2	42	87	135	232	312	490
3	42	87	136	232		487
4	42	87	137	231		
5	42	87	136	232		
6	42	87	136	232		
7	42	87	137			
8	42	87	137			
9	42	87	138			
10	42	87	140			
Average	42	87	137	232	311	488

Table A.12b Dynamic, Memory Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	49	98	148	246	371	616
2	49	98	148	246	370	620
3	49	98	148	247	370	617
4	49	98	148	247	370	621
5	49	98	148	247	371	623
6	49	98	148	247	370	617
7	49	98	148	247	370	616
8	49	98	148	246	371	623
9	49	98	148	247	370	616
10	49	98	148	246	370	617
Average	49	98	148	247	370	619

Table A.13a Dynamic, Compute Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	7	59	2228	3647	5877
2	0	7	50	2229	3658	5886
3	0	7	75	2222		5867
4	0	7	75	2222		
5	0	7	49	2212		
6	0	7	62	2240		
7	0	7	75			
8	0	7	63			
9	0	7	72			
10	0	7	69			
Average	0	7	65	2226	3653	5877

Table A.13b Dynamic, Memory Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.14a Dynamic, Compute Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	61074	469677	36062036	54214137	79916099
2	0	58530	405521	36034164	54869624	80389375
3	0	58045	623957	36056563		80020200
4	0	58312	625016	35623670		
5	0	59550	385851	35632227		
6	0	60306	504305	36179428		
7	0	60097	605434			
8	0	58270	519204			
9	0	57002	583334			
10	0	57754	553924			
Average	0	58894	527622	35931348	54541881	80108558

Table A.14b Dynamic, Memory Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.15a Dynamic, Compute Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	405499	2456007	85793291	202815481	388229384
2	0	386482	2261308	86419114	201375436	387557106
3	0	383907	2805812	85995835		387843197
4	0	384221	2834225	86318483		
5	0	396090	2169293	86362851		
6	0	399497	2589244	86638797		
7	0	399401	2752834			
8	0	386451	2623343			
9	0	377142	2668057			
10	0	383647	2664954			
Average	0	390234	2582508	86254729	202095459	387876562

Table A.15b Dynamic, Memory Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

A.4 Automatic Allocation

Table A.16a Automatic, Compute Node: Process Size (GB) vs Execution Time (s)

Trial	4G	8G	12G	20G	30G	50G
1	42	161	2255	172041	216093	261728
2	42	167	2537			
3	42	168	1148			
4	42	165	2287			
5	42	166	1218			
6	42	165	1830			
7	42	166	1190			
8	42	166	1835			
9	42	169	2535			
10	42	168	1127			
Average	42	166	1796	172041	216093	261728

Table A.16b Automatic, Memory Node: Process Size (GB) vs Execution Time (s)

Trial	4	8	12	20	30	50
1	49	99	148	246	371	624
2	49	99	147	247	373	623
3	49	98	147	247	371	624
4	49	98	147	246	368	621
5	49	98	147	247	370	624
6	50	98	148	246	371	625
7	49	98	148	246	370	624
8	49	98	148	247	370	619
9	49	98	147	247	370	624
10	49	98	147	247	368	620
Average	49	98	147	247	370	623

Table A.17a Automatic, Compute Node: Process Size (GB) vs User Time (s)

Trial	4G	8G	12G	20G	30G	50G
1	42	87	136	232	312	489
2	42	87	137			
3	42	87	136			
4	42	87	136			
5	42	87	136			
6	42	87	135			
7	42	87	136			
8	42	86	136			
9	42	87	136			
10	42	87	136			
Average	42	87	136	232	312	489

Table A.17b Automatic, Memory Node: Process Size (GB) vs User Time (s)

Trial	4	8	12	20	30	50
1	49	99	148	246	371	624
2	49	99	147	247	373	623
3	49	98	147	247	370	624
4	49	98	147	245	368	621
5	49	98	147	247	370	624
6	50	98	148	246	371	624
7	49	98	148	246	370	623
8	49	98	148	247	370	619
9	49	98	147	247	370	624
10	49	98	147	246	368	620
Average	49	98	147	246	370	623

Table A.18a Automatic, Compute Node: Process Size (GB) vs System Time (s)

Trial	4G	8G	12G	20G	30G	50G
1	0	6	66	2210	3650	5938
2	0	7	74			
3	0	7	49			
4	0	7	69			
5	0	7	50			
6	0	7	62			
7	0	7	50			
8	0	7	62			
9	0	7	75			
10	0	7	49			
Average	0	7	61	2210	3650	5938

Table A.18b Automatic, Memory Node: Process Size (GB) vs System Time (s)

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.19a Automatic, Compute Node: Process Size (GB) vs Page Faults

Trial	4G	8G	12G	20G	30G	50G
1	0	59024	571327	35434342	54329159	81590868
2	0	59192	603975			
3	0	60208	393148			
4	0	59119	567007			
5	0	59041	400538			
6	0	58637	504817			
7	0	59782	396176			
8	0	59961	497012			
9	0	62114	611398			
10	0	60042	383922			
Average	0	59712	492932	35434342	54329159	81590868

Table A.19b Automatic, Memory Node: Process Size (GB) vs Page Faults

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Table A.20a Automatic, Compute Node: Process Size (GB) vs Page Reclaims

Trial	4G	8G	12G	20G	30G	50G
1	0	384064	2802431	86225746	202455257	386017177
2	0	381232	2761805			
3	0	396342	2229176			
4	0	389252	2756720			
5	0	388532	2276499			
6	0	385009	2583701			
7	0	394245	2231233			
8	0	394775	2585920			
9	0	411704	2730812			
10	0	395031	2206420			
Average	0	392019	2516472	86225746	202455257	386017177

Table A.20b Automatic, Memory Node: Process Size (GB) vs Page Reclaims

Trial	4	8	12	20	30	50
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
Average	0	0	0	0	0	0

Curriculum Vitae

Jon Ramirez was born on September 21, 1984 in El Paso, Texas. The first born of Jose and Phuong Ramirez, he graduated from Americas High School, El Paso, Texas in 2003. He entered California Institute of Technology in Pasadena, California, where he earned a bachelor's of science degree in electrical engineering in the spring term of 2007. In the fall of 2007, he entered the computer engineering master's of science degree program at University of Texas, El Paso. While he pursued his master's degree at UTEP, Jon served as a system administrator at the Electrical and Computer Engineering Department's UNIX lab and was active within the Electrical and Computer Engineering's Distributed Computing Lab research group.