ALNEELAIN UNIVERSITY
GRADUATE COLLEGE

Cluster Infrastructure & Computing
Building Local Cluster Environments

A thesis
Submitted for the requirements of degree of Master of Science (M.sc) Electronic & Computational Electronic

By
Ibrahim Elhaj Ali Mohamed

Supervisor
Prof. Abdel Monim Artoli

2012
مستقبل البحث

يهدف الباحث من خلال هذه الامثلة البحث في بيئة الحوسبة المتوازية من أجل تحسين الإداء وتحقيق الاستخدام الأمثل والاستفادة القصوى من العداد الحاسوبي لدعم البحوث والتجارب العلمية بوحدة الحوسبة بمدرسة الفيزياء والفيزياء التطبيقية بكلية العلوم والتكنولوجيا جامعة النيلين. وذلك من خلال بناء بيئة حاسوبية منخفضة التكلفة من اجهزة الكمبيوتر المتوفرة لمساعدة الاجهزة الكمبيوتر المركزية فائقة الإداء التي لا توفر الغير متوفر لدينا بسبب القيود التكنولوجية على بانانا.

من خلال هذه الامثلة درس الباحث طرق انشاء العناقيد الحاسوبية وطرق اختبارها، علاوة على ذلك تم انشاء عناقيد حاسوبية قابلة للزيادة مكونه من اربع حواسيب ذات معالج انتل بنتيوم 4 ثم اجريت عليها اختبارات باستخدام المعايير القياسية التي طورت OSU Micro-Benchmarks بواسطة وكالة ناسا وجامعة أوهايو وهي (NPB3.3). و伎 3.1 على التوالي وكذلك برنامج ضرب مصفوفتين باستخدام واجهة تمرير الرسال (MPI) لقياس زمن الكمون والذي رصد بين 48.95 - 94.3444484 ميكرو ثانية وعرض الحزمة والتي تراوحت بين 0.1 - 9.96 ميغابايت الثانيه.
Abstract

The researcher aimed through this thesis to investigate the parallel computing environment in order to enhance performance and achieve maximum efficient usage of computer hardware to support scientific research and scientific experiments the computational unit, School of Physics and applied physics in Faculty of Science and Technology at Alncclain University. The aim was to build a low cost computer environment based on available PCs to stand for the costly and unavailable large mainframe computers due to technological restrictions on our country. Through this thesis the researcher studied the methodology of creating computer clusters and their testing means. Furthermore, a possible extendable and incremental cluster of four nodes was created using Intel Pentium 4 processor. Performance tests were run using a standard benchmark developed by NASA and Ohio State University namely NPB3.3 and OSU Micro-Benchmarks 3.5.1 respectively. Message Passing Interface (MPI) Matrix Multiplication $A*B = C$ Parallel benchmark code, was implemented to capture latency which was found to range from 48.95 to 844447.34 micro second and Bandwidth which was found to range 0.01 to 9.96 MB/s.
ACKNOWLEDGEMENTS

First and foremost, a credit goes to my father, mother, and the rest of
my family for their unlimited and generous support.

I would like to take this opportunity to thank Prof. Abdel Monim A
rtoli for his proactive supervision for this research and for me
personally in my academic stepping.

I would like also to thank the family of School of Physics and
applied physics in Faculty of Science and Technology at Alneclain
University.
For all help that I found from them.

Great thank for my entire friend for their help and all who dedicated
and supported me through this stage of my life.
Arabic abstract

Abstract

ACKNOWLEDGEMENTS

Contents

Chapter one
1.1 Introduction 1
1.2 Previous studies 2
1.3 Thesis outline 3

Chapter Two
2 Theoretical Background 4
2.1 Background in parallel architecture 4
2.2 Multicore processors 4
2.2.1 Hierarchical design 5
2.2.2 Pipelined design 6
2.2.3 Network-based design 8
2.3 The development of multicore processor 10
2.3.1 Dual-core processor 10
2.3.1.1 AMD Dual-core chips 11
2.3.1.2 Intel Dual-core chips 12
2.3.2 Quad-core processors 13
2.4 Parallel architecture 14
2.4.1 Single Instruction Single Data (SISD) 15
2.4.2 Multiple Instruction Single Data (MISD) 15
2.4.3 Single Instruction Multiple Data (SIMD) 15
2.4.4 Multiple Instruction Multiple Data (MIMD) 16
2.5 Memory organization of parallel computer
   2.5.1 Computer with distributed memory organization
   2.5.2 Computer with shared memory organization
2.6 Interconnection network
   2.6.1 Topology
   2.6.2 Static interconnection network
   2.6.3 Dynamic interconnection network
   2.6.4 Routing technique
2.7 Summary

**Chapter Three**

3.1 Building local Cluster Environment
3.2 Requirements for building a cluster
3.3 Our Cluster Components
   3.3.1 Nodes
   3.3.2 Network for platform
   3.3.3 Platform Software
3.4 The architecture of our cluster system
3.5 Installation Procedures
   3.5.1 Installing Linux and configuring nodes
   3.5.2 Configures the NFS server
   3.5.3 Installing and Running LAM
   3.5.4 Compiling and Executing MPI programs
3.6 MPI (Message Passing Interface)
   3.6.1 MPI Program Structure
   3.6.2 A Generic MPI Program
   3.6.3 MPI Header Files
Chapter five

5 Results and analysis
5.1 Introduction
5.2 Machine Data
5.2.1 Reference Results Summary for class A
5.3 Our Machine Data
5.3.1 Benchmark result to our cluster according to A package size
5.3.9 The summarized results
5.3.10 Graphically representation of the result
5.4 Reference Summary Results for class B
5.4.1 Graphical representation of reference result Class B
5.5 Class B Benchmark results in accordance to our architecture
5.6 The summarized results
5.6.1 Graphical representation of Class B Benchmark results:
5.7 Comparisons between study and reference
5.7.1 Class A
5.7.2 Class B
5.8 Summary of the comparisons
5.8.1 For class A packet size
5.8.2 For Class B packet size
5.9 Latency and bandwidth
5.9.1 Latency and bandwidth result
5.9.2 Graphical representation of latency and bandwidth
5.10 Speed scaling of our cluster
Chapter Six

Conclusion and future work 94

6.1 Conclusion 94

6.2 Recommendation 95

References 96
List of Figures

Figure 2.1 Hierarchical Design 6
Figure 2.2: Pipelined Designs 7
Figure 2.3: Xelerator X11 network processor as an example for a 8 pipelined design
Figure 2.4: Network-Based Design 9
Figure 2.5: Intel Teraflop processor (a network-based design of a 10 multicore processor)
Figure 2.6: AMD Athlon 64X2 11
Figure 2.7: Intel Core 2 Duo E6400 12
Figure 2.8 AMD Quad-core Opteron 13
Figure 2.9 Intel Quad core Xeon 14
Figure 3.1 Typical Node ports description 23
Figure 3.2: The architecture of simple cluster system 25
Figure 3.3: interconnection and the developed cluster 26
Figure 3.4: hosts file 30
Figure 3.5: services window 31
Figure 3.6: export file 31
Figure 3.7: fstab file 32
Figure 3.8: lam-bhost.def file 33
Figure 3.9: terminal run lam 33
Figure 3.10: MPI program structure 35
Figure 3.11 output MPI code 47
Figure 5.1: class A Execution Time accordance 53
Figure 5.2: Class A Total Mega Flops for 4 nps 54
Figure 5.3: Class A Mega process per CPU 54
List of Tables

Table 3.1: partition desk space ...................................... 28
Table 3.2 C data type used in MPI .................................. 38
Table 3.3 Fortran data type used in MPI ......................... 39
Table 5.1: data of reference node ..................................... 52
Table 5.2: summary of reference result class A np 4 ............. 53
Table 5.3: data of cluster node ....................................... 55
Table 5.4: SP Benchmark result ....................................... 55
Table 5.5: LU Benchmark result ....................................... 56
Table 5.6: IS Benchmark result ....................................... 56
Table 5.7: FT Benchmark result ....................................... 57
Table 5.8: EP Benchmark result ....................................... 57
Table 5.9 CG Benchmark result ....................................... 58
Table 5.10 BT Benchmark result ...................................... 58
Table 5.11 MG Benchmark result ...................................... 59
Table 5.12: result summary of cluster ............................... 59
Table 5.13: summary of reference result class B np 4 .......... 61
Table 5.14: SP Benchmark result ...................................... 63
Table 5.15: MG Benchmark result ...................................... 64
Table 5.16: LU Benchmark result ...................................... 64
Table 5.17: IS Benchmark result ...................................... 65
Table 5.18: FT Benchmark result ...................................... 65
Table 5.19: EP Benchmark result ...................................... 66
Table 5.20: CG Benchmark result ...................................... 66
Table 5.21: BT Benchmark result ...................................... 67
Table 5.22: Summarized result ........................................ 67
Table 5.23: A. Execution Time analysis
Table 5.24: A. Total Megaflop
Table 5.25: A Megaflop per processes
Table 5.26: Execution Time for class B
Table 5.27: Total Megaflop for class B
Table 5.28: Megaflop per processor for class B
Table 5.29: Result of OSU benchmark
Table 5.30: summarizing the MPI code result
1.1 Introduction

A computing infrastructure is highly needed in many engineering, educational and research problems. Lack of computing power in the Sudan is witnessed. To our knowledge, there is no computing infrastructure that is capable of facing the extensive computing demand problems. Therefore, computer researchers introduced a new notation, which is parallel processing that refers to the concept of speeding-up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor. A program being executed across n processors might execute faster than it would using a single processor. The aim of this study is to describe and implement an infrastructure for relatively high performance computing in the department of physics to help in responding to the demands for high speed heavy computing within the Computational Science Group in Alneelain University.

Traditionally, multiple processors were provided within specially designed “parallel computer”, along these lines. Clustering is a powerful concept and technique for deriving capabilities from existing classes of components. In nature, clustering is a fundamental mechanism for creating complexity and diversity through the aggregation and collection of simple basic elements. To accomplish such collection, an intervening medium of combination and exchange is required that establishes the interrelationships among the constituent elements and facilities, their cooperative interactions from which is derived the emergent behavior of the compound entity.

In the field of computing systems, clustering is being applied to provide new system structures from existing computing elements to deliver
capabilities that through other approaches could easily cost ten times as much. In recent years clustering hardware and software have evolved so that today potential user institutions have a number of choices in terms of form, scale, environments, cost, and means of implementation to meet their scalable computing requirements. But clusters are also playing important roles in medium scale technical and commerce computing, taking advantage of low-cost, mass-market pc-based computer technology. These so called Beowulf-class systems have become extremely popular, providing exceptional price/performance, flexibility of configuration and scalability to provide a powerful new tool, opening up entirely new opportunities for computing applications.

However, the running cost and technical support prevent these clusters from being widespread in the Middle East. Perhaps vector computing is better option if a small budget of put less than look's can be secured.

1.2 Previous studies

Cluster and heavy computing facilities were demanded in University research cluster. In Alneelain university, Parsytec vector was existing at the physics department since 2000, but never use due to lack of technical support and network infrastructure, in addition to lack of finance.

A number of trials to build cluster in Sudan university of science and technology and university of Khartoum were reported [Umran, Group of Usama Rayes, Anas]

However during the implementation of the stud, our objective was to provide necessary computing power to the computational group for DMRG
Clad CFD calculations when it was not possible to have enough computing resource, the researchers traveled outside to German and Italy to finalize their research [Anas, Walead].

1.3 Thesis outlines

Chapter two gives theoretical background for a parallel architectures and the theory of parallelism. Materials and method are explained in chapter three in chapter four discusses the benchmark and the result presented and analysis given in chapter five in chapter six conclusion with recommendations for future work finally references are given at the end of the thesis.
Chapter Two
2. Theoretical Background

2.1 Background in Parallel Architecture

The processor chips are the most important components in the computer. The processor contains a number of transistors to design the functional units of the processor. The number of transistors doubles every 18-24 months according to Moore’s law (Jack Dongarra, 2003). This increase in number of transistors used to improve the architecture of processor, like adding functional units, more cache and more registers. This enables hardware manufacturers to provide a significant performance increase for application programs. But still no major enhancement occurs with respect to multi-processing putting in mind the exception of the pipelining technique which introduced the effective and efficient use of time sharing notion utilizing an architecture composed of more than a processor in the CPU unit (Thomas Rauber, 2010).

2.2 Multicore Processors Architecture

Starting in the 1990s, the increasing capacity of a single chip allowed designers to place multiple processors on a single die. This approach, initially called on-chip multiprocessing or single-chip multiprocessing, has come to be called multicore. Multicore processors integrate execution cores on a single multiple processor chip. For the operating system, each execution core represents an independent logical processor with separate execution resources like functional units or execution pipelines. Each core has to be
controlled separately, and the operating system can assign different application programs to the different cores to obtain a parallel execution. Background applications like virus checking, image compression and encoding can run in parallel to application programs of the user. By using techniques of parallel programming, it is also possible to execute a computation-intensive application program (like computer games, computer vision, or scientific simulations) in parallel on a set of cores, thus reducing the execution time compared to an execution on a single core or leading to more accurate results by performing more computations as in the sequential case.

The use of multiple cores on a single processor chip also enables standard programs, like text processing, office applications, or computer games, to provide additional features that are computed in the background on a separate core so that the user does not notice any delay in the main application. But again, techniques of parallel programming have to be used for the implementation. (Thomas Rauber, 2010).

There are many types of architectures to design multicore processor, according to number of cores, size of the cache and access of cores to cache. Architectural design aspects can be categorized as follows:

2.2.1 Hierarchical Design

For a hierarchical design, multiple cores share multiple caches. The caches are organized in a tree-like configuration, and the size of the cache increase from the leaves through the root. The root represents the connection to external memory. Thus, each core can have a separate L1 cache and shares the L2 cache with other cores. All cores share the common external memory.
This can be extended to more levels. Additional sub-components can be used to connect the caches of one level with each other. A typical usage area for a hierarchical design is the symmetric multiprocessing (SMP) configuration. A hierarchical design is also often used for standard desktop or server processors. Examples are the IBM Power6 architecture, the processors of the Intel Xeon and AMD Opteron family, as well as the Sun Niagara processors (T1 and T2) (Thomas Rauber 2010).

![Hierarchical Design Diagram]

**Figure 2.1: Hierarchical Design**

### 2.2.2 Pipelined Designs

For a pipelined design, data elements are processed by multiple execution cores in a pipelined way. Data elements enter the processor chip via an input port and are passed successively through different cores until the processed data elements leave the last core and the entire processor chip via an output port. Each core performs specific processing steps on each data element.
Network processors used in routers and graphics processors both perform this style of computations. (Thomas Rauber, 2010).

![Figure 2.2: Pipelined Designs](image)

Network processors which are used in routers and graphics processors apply this computational concept. Network processors samples with a pipelined design are the Xelerator X10 and X11 processors for the successive processing of network packets in a pipelined way within the chip. The Xelerator X11 contains up to 800 separate cores which are arranged in a logically linear pipeline. The network packets to be processed enter the chip via multiple input ports on one side of the chip, are successively processed by the cores, and then exit the chip.
Figure 2.3: Xelerator X11 network processor as an example for a pipelined design

2.2.3 Network-Based Design

For a network-based design, the cores of a processor chip and their local caches and memories are connected via an interconnection network with other cores of the chip. Data transfer between the cores is performed via the interconnection network. This network may also provide support for the synchronization of the cores. Off-chip interfaces may be provided via specialized cores or DMA ports. (Thomas Rauber, 2010).
The Teraflop processor developed as a prototype contains 80 cores, which are arranged in a 8×10 mesh. Each core can perform floating-point operations and contains a local cache as well as a router to perform data transfer between the cores and the main memory. There are additional cores for processing video data, encryption, and graphics computations. Depending on the application area, the number of specialized cores of such a processor chip could be varied.
Figure 2.5: Intel Teraflop processor (a network-based design of a multicore processor)

2.3 The development of multicore processor

2.3.1 Dual-core processor

A dual core processor is a CPU with two separate cores sharing the same chip, each with its own cache. In a dual core processor each core handles incoming data strings consecutively improving efficiency. Now when one is executing the other can be accessing the system bus or executing its own code. Implementing this concept, AMD and Intel's dual-core flagships are 64-bit. <http://www.seminarpaper.com/2011/09/dual-core-processor.html>
2.3.1.1 AMD Dual-Core Chips

In April 2005, AMD delivered its first dual-core chip to the computer market, the Opteron processor. There are several Opteron dual-core 90nm processors ranging in clock speeds from 1.6GHz to 2.4GHz. The first desktop dual-core processor from AMD appeared in May 2005 under the Athlon 64 X2 brand names.

![Diagram of AMD Athlon 64X2 architecture](image)

**Figure 2.6: AMD Athlon 64X2**

The Athlon 64X2 is designed specifically for multiple cores in a single chip (Figure). It also employs private L2 caches, both L2 caches share a system request queue, which connects with an on-die memory controller and a HyperTransport. The HyperTransport removes system bottlenecks by reducing the number of buses required in a system

2.3.1.2 Intel Dual-Core Chips

Intel announced its first dual-core processor architecture in 2005. The two cores share the same functional execution units and cache hierarchy; however, the OS recognizes each execution core as an independent processor.

![Diagram of Intel Core 2 Duo E6400](image)

**Figure 2.7: Intel Core 2 Duo E6400**

The Intel Core 2 Duo (Figure 2.7) E6400 emphasizes mainly on cache efficiency and does not stress on the clock frequency for high power efficiency, the Core 2 Duo processor has more ALU units. Core 2 Duo employs a shared L2 cache to increase the effective on-chip cache capacity.
Upon a miss from the core’s L1 cache, the shared L2 and the L1 of the other core are looked up in parallel before sending the request to the memory. The cache block located in the other L1 cache can be fetched without off-chip traffic.

2.3.2 Quad-core processor

A quad core processor is a single chip content four independent cores. Each of the four cores works independently to read and execute computer program orders, which might contain data and memory functions. Quad core processors in computer systems are usually designed toward users who run resource intensive applications such as video games, and graphic editors. Many video games are written so that they make optimal use of multi-processors. Both AMD and Intel product Quad-core processor like AMD Quad-core opteron and Quad-core Intel Xeon which are showing in the figure below

Figure 2.8 AMD Quad-core Opteron
2.4 Parallel Architectures

A parallel computer can be characterized as a collection of processing elements that can communicate and cooperate to solve large problems fast. (Thomas Rauber, 2010). This definition is intentionally quite vague to capture a large variety of parallel platforms. Many important details are not addressed by the definition, including the number and complexity of the processing elements, the structure of the interconnection network between the processing elements, the coordination of the work between the processing elements, as well as important characteristics of the problem to be solved. Flynn’s taxonomy characterizes parallel computers into four categories, according to the global control and the resulting data and control flows.
2.4.1 Single-Instruction, Single-Data (SISD)

There is one processing element which has access to a single program and data storage. In each step, the processing element loads an instruction and the corresponding data and executes the instruction. The result is stored back in the data storage. Thus, SISD is the conventional sequential computer according to the von Neumann model (Thomas Rauber, 2010).

2.4.2 Multiple-Instruction, Single-Data (MISD)

There are multiple processing elements each of which has a private program memory, but there is only one common access to a single global data memory. In each step, each processing element obtains the same data element from the data memory and loads an instruction from its private program memory. These possibly different instructions are then executed in parallel by the processing elements using the previously obtained (identical) data element as operand. This execution model is very restrictive and no commercial parallel computer of this type has ever been built (Thomas Rauber, 2010).

2.4.3 Single-Instruction, Multiple-Data (SIMD)

There are multiple processing elements each of which has a private access to a (shared or distributed) data memory, for a discussion of shared and distributed address spaces. But there is only one program memory from which a special control processor fetches and dispatches instructions. In each step, each processing element obtains from the control processor the same
instruction and loads a separate data element through its private data access on which the instruction is performed. Thus, the instruction is synchronously applied in parallel by all processing elements to different data elements. For applications with a significant degree of data parallelism, the SIMD approach can be very efficient. Examples are multimedia applications or computer graphics algorithms to generate realistic three-dimensional views of computer-generated environments. (Thomas Rauber, 2010).

2.4.4 Multiple-Instruction, Multiple-Data (MIMD)

There are multiple processing elements each of which has a separate instruction and data access to a (shared or distributed) program and data memory. In each step, each processing element loads a separate instruction and a separate data element, applies the instruction to the data element, and stores a possible result back into the data storage. The processing elements work asynchronously with each other. Multicore processors or cluster systems are examples for the MIMD model. (Thomas Rauber, 2010).

2.5 Memory Organization of Parallel Computers

Parallel computers are based on the MIMD model which classification can be done according to their memory organization and there is two aspects can be distinguished memory organization, the physical memory organization and the view of the programmer of the memory. (Thomas Rauber, 2010).
2.5.1 Computers with Distributed Memory Organization

Computers with a physically distributed memory consist of a number of processing elements (called nodes) and an interconnection network which connects nodes and supports the transfer of data between nodes. A node is an independent unit, consisting of processor, local memory. Program data is stored in the local memory of one or several nodes. All local memory is private and only the local processor can access the local memory directly. When a processor needs data from the local memory of other nodes to perform local computations, message-passing has to be performed via the interconnection network. Therefore, distributed memory machines are strongly connected with the message-passing programming model which is based on communication between cooperating sequential processes and the topology of the communication layout between the nodes if of extreme importance in establishing fast connections between the nodes. (Thomas Rauber, 2010).

2.5.2 Computers with Shared Memory Organization

Computers with a physically shared memory consist of a number of processors or cores, a shared physical memory (global memory), and an interconnection network to connect the processors with the memory. The shared memory can be implemented as a set of memory modules. Data can be exchanged between processors via the global memory by reading or writing shared variables. The cores of a multicore processor are an example for a shared memory computer. Physically, the global memory usually
consists of separate memory modules providing a common address space which can be accessed by all processors. (Thomas Rauber, 2010).

2.6 Interconnection Networks

Similar to control flow and data flow, or memory organization, the interconnection network can also be used for a classification of parallel systems. Internally, the network consists of links and switches which are arranged and connected in some regular way. In multi computer systems, the interconnection network is used to connect the processors or nodes with each other. Interactions between the processors for coordination, synchronization, or exchange of data are obtained by communication through message-passing over the links of the interconnection network. In multiprocessor systems, the interconnection network is used to connect the processors with the memory modules. Thus, memory accesses of the processors are performed via the interconnection network.
In both cases, the main task of the interconnection network is to transfer a message from a specific processor to a specific destination. The message may contain data or a memory request. The destination may be another processor or a memory module. The requirement for the interconnection network is to perform the message transfer correctly as fast as possible, even if several messages have to be transferred at the same time. Message transfer and memory accesses represent a significant part of operations of parallel systems with a distributed or shared address space. Therefore, the interconnection network used represents a significant part of the design of a parallel system and may have a large influence on its performance. Important design criteria of networks are (Thomas Rauber, 2010).

2.6.1 Topology

The topology of an interconnection network describes the geometric structure used for the arrangement of switches and links to connect processors or processors and memory modules. The geometric structure can be described as a graph in which switches, processors, or memory modules are represented as vertices and physical links are represented as edges. It can be distinguished between static and dynamic interconnection networks. (Thomas Rauber, 2010).

2.6.2 Static interconnection networks

In the static network nodes (processors or memory modules) connected directly with each other by fixed physical links. They are also called direct
networks or point-to-point networks. The number of connections to or from a node may vary from only one in a star network to the total number of nodes in the mesh network for a completely connected graph. Static networks are often used for systems with a distributed address space where a node comprises a processor and the corresponding memory module. (Thomas Rauber, 2010).

2.6.3 Dynamic interconnection networks

In the dynamic network nodes connect indirectly via switches and links. They are also called indirect networks. Examples of indirect networks are bus-based networks or switching networks which consist of switches connected by links. Dynamic networks are used for both parallel systems with distributed and shared address space. Often, hybrid strategies are used. (Thomas Rauber, 2010).

2.6.4 Routing technique

The routing technique determines how and along which path messages are transferred in the network from a sender to a receiver. A path in the network is a series of nodes along which the message is transferred. Important aspects of the routing technique are the routing algorithm which determines the path to be used for the transmission and the switching strategy which determines whether and how messages are cut into pieces, how a routing path is assigned to a message, and how a message is forwarded along the processors or switches on the routing path. The combination of routing
algorithm, switching strategy, and network topology determines the performance of a network. (Thomas Rauber, 2010).

2.7 Summary

We could summarize what the researcher had introduced here, by firstly giving a full background for parallelism and multicore processor architecture, stating the Intel and AMD processors architecture with regard to the Dual Core and Quad Core processors, descending to the different processor instructions and data sets (SISD, SIMD, MISD and MIMD). Addressing the memory organization was part of what the researcher investigated introducing the shared and distributed memory. Finally, internetworking techniques were investigated.
Chapter Three
3.1 Building local Cluster Environment

Clustering is the use of multiple computers, multiple storage devices, and redundant interconnections, to form what appears to users as a single highly available system. Cluster computing can be used for load balancing as well as for high availability. It is used as a relatively low-cost form of parallel processing machine for scientific and other applications that lend themselves to parallel operations.

Clusters of systems or workstations, is a type of parallel or distributed computing system that consists of a master node which provides cluster management services and one or more computing nodes.

3.2 Requirements for building a cluster

The basic requirement for building a cluster of workstation is addressed as follows:

First: you need to collect multiple PCs and select one PC as a master node and other PCs as slave nodes. The master node requires at least two Ethernet network cards. One card serves as a gateway with a static IP address for outer connection to the Internet. The other card connect the master node with other nodes in the cluster. For security reasons, only master node is allowed to communicate to the outside network. Connection between an individual slave node and an outside network is not recommended

Second: a network media, to group all computer nodes together into a cluster, in which the nodes can communicate with each other during parallel computing.

Third: operating system and software to run the parallel application.
3.3 Our Cluster Components

3.3.1 Nodes

We have chosen hp Compaq dx 6100 as our node in the cluster, with the following architecture:

*Genuine Intel Pentium® Pentium 4 CPU speed 3.0 GHz*

*Cache 1024 KB*

*Local memory 1GB*

*Network card Integrated Broadcom NetXtreme Gigabit Ethernet NIC*

*And 3com as the other network card in the master node*

![Figure 3.1 Typical Node ports description](image-url)
3.3.2 Network for platform

In the network we build local network using switch 3COM-3300 as our inter-connection switch, RJ45 and UTP cat5e cable to connect the nodes to the switch.

3.3.3 Platform Software

For the operating system we used Red Hat Enterprise 4 Linux. Clustering can be performed on various operating systems like Windows, Macintosh, Solaris etc., we used Linux for its advantages which are as follows:

- Linux runs on a wide range of hardware
- Linux is exceptionally stable
- Linux source code is freely distributed.
- Linux is relatively virus free.
- Having a wide variety of tools and applications for free.
- Good environment for developing cluster infrastructure

As for the parallel software packages we used Lam-MPI package.
3.4 The architecture of our cluster system

Figure 3.2: The architecture of simple cluster system
Figure 3.3: Interconnection and the developed cluster
3.5 Installation Procedures

3.5.1 Installing Linux and configuring nodes

Many methods exist to install LINUX on PCs. One method we used is to boot each PC by using bootable CD of LINUX operating system.

I. Installs the LINUX on our nodes

The step-by-step guidance of installing LINUX on PC is listed below.

1. We insert the Red Hat Enterprise 4 LINUX CD#1 into the CD-ROM drive of the master node.
2. Restart or reboot the node. It should automatically boot your computer. If it doesn’t, we need to check our PC’s initial setting (Press F1 or Delete key or F2 key to enter the BIOS mode, when we start our PC; we check our computer’s specification for details).

We make the computer has its initial set-up, which allows CD-ROM to boot the system.

3. After booting our PC, the screen was prompt a welcome message. Press the Enter key to choose a graphical user interface (GUI) for installation.
4. We select English as the installation language.
5. We select default keyboard configuration.
6. We select default mouse configuration.
7. Choose the type of installation. Red Hat enterprise 4 LINUX provides five installation types. They are Workstation, Server, Laptop, Custom and Update. In the present study, we chose the Custom type to install Red Hat LINUX on the master and the slave nodes respectively, in order to precisely
configure our cluster system for better understanding of the installation process.

8. Partitioning disk space in each node. The Red Hat’s Disk Druid partitioning tool had chosen. How to partition your hard drive disk totally depends on the customer’s opinion. Basically, the best PC was selected as our master node. Recommend that the master node have a high speed CPU, large memory, and a large disk space. The swap volume is usually recommended to be configured as double the installed memory volume. In the present study, the partition information for our master node is provided below as a reference. (Note: since the cluster system for demonstration hereby is a legacy system, Hp 6100 it our master node.)

Table 3.1: partition desk space

<table>
<thead>
<tr>
<th>Partition name</th>
<th>Mount point</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>/</td>
<td>250M</td>
</tr>
<tr>
<td>Boot</td>
<td>/boot</td>
<td>50M</td>
</tr>
<tr>
<td>Home</td>
<td>/home</td>
<td>6G</td>
</tr>
<tr>
<td>Usr</td>
<td>/usr</td>
<td>2.5G</td>
</tr>
<tr>
<td>Swap</td>
<td>2GB</td>
<td>(2*RAM)</td>
</tr>
</tbody>
</table>

The drive "/home" will be shared with other slave nodes via NFS.

9. Click Next button to select the default boot loader "GRUB".

10. Next step is to configure our Ethernet card of master node for networking. Red Hat LINUX will automatically find two Ethernet cards on the master node, named "eth0" and "eth1," respectively. For configuring "eth0" network, the option "Configure using DHCP" was selected and "Activate on boot." For configuring "eth1" network, the default option
"Activate on boot." was only selected. It is time for you to configure internal IP Address and host name. In the present study:

For master node:
IP Address: 192.168.0.1
Host Name: Master

For slave node
IP Address: 192.168.0.X
Host Names: node0X (where X is {1, 2, and 3} for each node)

11. The next step is to configure firewall security. On the master node the default security level "medium" had chosen. For the trusted device "eth0" the "DHCP" selected and "SSH." For the other trusted device "eth1" the "SSH." only selected.

On the slave node we just selected the protocol “SSH” for the trusted device “eth0”.

12. In the language support selection, "English" was chosen.

13. In the time zone configuration, "Africa/Sudan: Center Time" was chosen.


15. In authentication configuration, the default option selected.

16. In the package group selection, "KDE" and "GNOME" were chosen, respectively.

17. The checkbox “Select individual Packages”, which enable us to install some other protocol package on our cluster system, had to be activated.

18. It is important to note that for all nodes (master and slaves nodes) in our cluster system, “flat view” chosen to display all package alphabetically. The following packages, mainly "openssh," “openssh-server,” “openssh-client,”
"rsh," "rsh-server," and "xinetd" chosen. After such selections, RedHat LINUX will automatically play these functions.

19. Now Red Hat enterprise 4 LINUX installation for a LINUX cluster is completed.

20. login in the node with the user “root” to edits the file /etc/hosts for each node (master and slaves) (logging in with root user). In this study, the following entries are used.

127.0.0.1 Localhost
192.168.0.1 Master
192.168.0.2 node1
192.168.0.3 node2
192.168.0.4 node3

![Image](image.png)

Figure 3.4: hosts file

3.5.2 Configures the NFS server

1. Login in the node as a "root" user. In the LINUX console window, enter the command "setup". In the setup window which LINUX provides, "System Services" was chosen to active the following daemons: "network", "nfs", "nfslock", "portmap", "rsh", "rlogin", "sshd", and "xinetd". Note we select the item or unselect the item by the key of "Space".

30
2. In the master node editing the /etc/exports file to specify the file systems operations and facilities (to be shared), hosts (to be allowed) and the type of permissions (ro, rw). In this study, our "exports" file has the following entry
/home 192.168.0.1/24(rw, no_root_squash)
Note: (1) 192.168.0.1/24 presents the first 24 IP addresses, from 192.168.0.1 to 192.168.0.24 to access the exported file system.

(2) No space between 192.168.0.1/24 and (rw, no_root_squash)
In the slave node edit the file "/etc/fstab". In this study, a line on the bottom of this file was add to tell the current slave node the information about the remote host name, the shared directory, and the local mount directory.

```
master:/home /home nfs
```

![Fstab file](image.png)

Figure 3.7: fstab file

This approach enables slave nodes automatically and statically to mount the share directory of a remote node, whenever the machines were rebooting. The alternative method is to type the command "mount", which can be used to dynamically mount to the remote share directory without rebooting our machines.

For example: mount -t nfs master:/home /home

3. Add the new user on the node. Using the GUI tool "user manager" in the KDE or GNOME to add new user in the system.

4. Rebooting the nodes.
3.5.3 Installing and Running LAM

The latest version of LAM (rpm package) downloaded from the following link:  http://www.lam-mpi.org/7.0/download.ph
And install LAM using the following command
Rpm –Uvh lam*.rpm
In all nodes of the cluster, after that the lam hosts file was configured by editing /etc/lam/lam- bhosts – def which looks like the below screen:

![Image of lam-bhost.def file](image.png)

Figure 3.8: lam-bhost.def file

To run lam-mpi you must switch from the root user to the created user at each node of the cluster, because mpi cannot run in root user. After that run the command lamboot
3.5.4 Compiling and Executing MPI programs

Compilers for C, C++ and FORTRAN programs are mpicc, mpiCC (or mpic++) and mpif77 respectively. These compilers include all the relevant files and directories required for running the MPI programs.

Compiling the code by

mpicc -o file name.c file name.o
mpiCC -o file name.C++ file name.o

To run the code

mpirun -np 3 file name.o

3.6 MPI (Message Passing Interface)

In this section we give an overview and example to the Message Passing Interface (MPI). The message passing interface is a library of functions (in C) or subroutines (in FORTRAN) that you insert into source code to perform data communication between processes.

3.6.1 MPI Program Structure

The MPI program elements are described in detail for both C and FORTRAN, which includes:
- Header files
- MPI naming conventions
- MPI routines and return values
- MPI handles
- MPI data types
- Initializing and terminating MPI
- Communicators
- Getting communicator information: rank and size

3.6.2 A Generic MPI Program

All MPI programs have the following general structure showing in figure

![MPI Program Structure Diagram]

Figure 3.10: MPI program structure
3.6.3 MPI Header Files

MPI header files contain the prototypes for MPI functions/subroutines, as well as definitions of macros, special constants, and datatypes used by MPI. An appropriate "include" statement must appear in any source file that contains MPI function calls or constants.

In C the header file is

```c
#include <mpi.h>
```

In FORTRAN header file is

```fortran
INCLUDE 'mpif.h'
```

3.6.4 MPI Naming Conventions

The names of all MPI entities (routines, constants, types, etc.) begin with `MPI_` to avoid conflicts.

FORTRAN routine names are conventionally all upper case:

```fortran
MPI_XXXXX(parameter, ..., IERR)
```

Example: `MPI_INIT(IERR)`.

C function names have a mixed case:

```c
MPI_Xxxxx(parameter, ...)
```

Example: `MPI_Init(&argc, &argv)`.

The names of MPI constants are all upper case in both C and FORTRAN, for example;

```c
MPI_COMM_WORLD
ORLD, MPI_REAL, ...
```
In C, specially defined types correspond to many MPI entities. (In FORTRAN these are all integers.) Type names follow the C function naming convention above; for example; 

\textit{MPI\_Comm} is the type corresponding to an MPI "communicator".

### 3.6.5 MPI Routines and Return Values

MPI routines are implemented as functions in C and subroutines in FORTRAN. In either case generally an error code is returned, enabling you to test for the successful operation of the routine.

In C, MPI functions return an int, which indicates the exit status of the call.

\begin{verbatim}
int err;
...
err = MPI_Init(&argc, &argv);
...
\end{verbatim}

In FORTRAN, MPI subroutines have an additional INTEGER argument -- always the last one in the argument list -- that contains the error status when the call returns.

\texttt{INTEGER IERR}

\begin{verbatim}
...
CALL MPI_INIT(IERR)
...
\end{verbatim}

The error code returned is MPI\_SUCCESS if the routine ran successfully (that is, the integer returned is equal to the pre-defined integer constant MPI\_SUCCESS). Thus, you can test for successful operation with
In C:

```c
if (err == MPI_SUCCESS) {
...routine ran correctly...
}
```

In Fortran:

```fortran
if (IERR.EQ.MPI_SUCCESS) THEN
...routine ran correctly...
END IF
```

If an error occurred, then the integer returned has an implementation-dependent value indicating the specific error.

### 3.6.6 MPI Data types

In C, the basic MPI datatypes and their corresponding C types are

Table 3.2  C data type used in MPI

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>Float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>Double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
In FORTRAN, the basic MPI datatypes and their corresponding FORTRAN types are

Table 3.3 FORTRAN data type used in MPI

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>Fortran data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

3.6.7 Initializing MPI

The first MPI routine called in any MPI program must be the initialization routine MPI_INIT. This routine establishes the MPI environment, returning an error code if there is a problem.

MPI_INIT may be called only once in any program!

In C:

```c
int err;
...
err = MPI_Init(&argc, &argv);
```

In Fortran:

```
INTEGER IERR
...
CALL MPI_INIT(IERR)
```
3.6.8 Communicators

A communicator is a handle representing a group of processors that can communicate with one another. The communicator name is required as an argument to all point-to-point and collective operations.

- The communicator specified in the send and receive calls must agree for communication to take place.
- Processors can communicate only if they share a communicator.

There can be many communicators, and a given processor can be a member of a number of different communicators. Within each communicator, processors are numbered consecutively (starting at 0). This identifying number is known as the rank of the processor in that communicator.

- The rank is also used to specify the source and destination in send and receive calls.
- If a processor belongs to more than one communicator, its rank in each can (and usually will) be different.

MPI automatically provides a basic communicator called MPI_COMM_WORLD. It is the communicator consisting of all processors. Using MPI_COMM_WORLD, every processor can communicate with every other processor. You can define additional communicators consisting of subsets of the available processors.
3.6.9 Getting Communicator Information: Rank

A processor can determine its rank in a communicator with a call to MPI_COMM_RANK.
In C:

\texttt{int MPI_Comm_rank(MPI_Comm comm, int *rank);} 

In FORTRAN:

\texttt{MPI_COMM_RANK(COMM, RANK, IERR)}

3.6.10 Getting Communicator Information: Size

A processor can also determine the size, or number of processors, of any communicator to which it belongs with a call to MPI_COMM_SIZE.
In C:

\texttt{int MPI_Comm_size(MPI_Comm comm, int *size);} 

- The argument comm is of type MPI_Comm, a communicator
In FORTRAN:

\texttt{MPI_COMM_SIZE(COMM, SIZE, IERR)}

3.6.11 Terminating MPI

The last MPI routine called should be MPI_FINALIZE which

- Cleans up all MPI data structures, cancels operations that never completed,
- Must be called by all processes; if any one process does not reach this statement, the program will appear to hang.
In C:

```c
int err;
...
err = MPI_Finalize();
```

In FORTRAN:

```fortran
INTEGER IERR
...
call MPI_FINALIZE(IERR)
```

### 3.7 An example in c

```c
#include <stdio.h>
#include <mpi.h>
int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n", my_rank, size);
  MPI_Finalize ();
  return 0;
}
```

### 3.7.1 Output

```
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
Hello world, I've rank 0 out of 4 procs.
```
3.8 Other example of mpi

DESCRIPTION: MPI timing example code. C version. In this example code, an MPI communication timing test is performed. The processor with mypid = 0 will send integer messages of "length" elements to the processor with mypid = 1 "REPS" times. Upon receiving the message a message of identical size is sent back. Before and after timings are made for each rep and an average calculated when completed. Also the Bandwidth is calculated.

#include "mpi.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
define REPS 1000
define MAXLENGTH 8000000

int main(argc,argv)
int argc;
char *argv[];
{
    int i,n,length;
    int *inmsg, *outmsg;
    int mypid,mysize;
    int rc;
    double start,finish,time;
    double bw;
    MPI_Status status;


/* Initialize MPI */
rc = MPI_Init(&argc,&argv); /* Get the size of the
MPI_COMM_WORLD communicator group */
rc|= MPI_Comm_size(MPI_COMM_WORLD,&mysize);
    /* Get my rank in the MPI_COMM_WORLD communicator
group */
rc|= MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if (mysize != 2)
{
    fprintf(stderr, "Error: Set environment variable MP_PROCS
to 2\n");
    exit(1);
}
length = 1;
inmsg = (int *) malloc(MAXLENGTH*sizeof(int));
outmsg = (int *) malloc(MAXLENGTH*sizeof(int));
    /* synchronize the processes */
rc = MPI_Barrier(MPI_COMM_WORLD);
    /* Task 0 processing */
if (myid == 0)
{
    for (i=1; i<=4; i++)
    {
        time = 0.00000000000000;
        /* round-trip timing test */
        printf("\n\nDoing round trip test for:\n");
        printf("Message length = %d integer
value(s)\n",length);
        printf("Message size = %d Bytes\n",4*length);
        printf("Number of Reps = %d\n",REPS);
        start = MPI_Wtime();
        for (n=1; n<=REPS; n++)

44
{ 
    /* send message to process 1 */
    rc=MPI_Send(&outmsg[0], length, MPI_INT, 1, 0, MPI_COMM_WORLD); 
    /* Now wait to receive the echo reply from process 1 */
    rc=MPI_Recv(&inmsg[0], length, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
}

finish = MPI_Wtime();
/* Calculate round trip time average and bandwidth, and print */
    time = finish - start;
    printf("*** Round Trip Avg = %f uSec\n", time/REPS);
    bw = 2.0*REPS*4.0*length/time;
    printf("*** Bandwidth = %f Byte/Sec\n", bw);
    printf(" Megabit/Sec\n", bw*8.0/1000000.0);
    length = 100*length;
}

/* Task 1 processing */
if (mypid == 1)
{
    for (i=1; i<=4; i++)
    {
        for (n=1; n<=REPS; n++)
        {
            /* receive message from process 0 */
            rc=MPI_Recv(&inmsg[0], length, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            /* return message to process 0 */
            rc=MPI_Send(&outmsg[0], length, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }
}
length = 100*length;
}
}

/* Finalize MPI */
MPI_Finalize();
exit(0);
}
3.8.1 Output

```
[ibrahim@master ~/]S: mpirun -o -l tim

Doing round trip test for:
Message length = 1 integer value(s)
Message size = 4 Bytes
Number of Reps = 1000
*** Round Trip Avg = 0.000111 uSec
*** Bandwidth = 72100 6822654 Byte/Sec
= 0.576805 Megabit/Sec

Doing round trip test for:
Message length = 100 integer value(s)
Message size = 400 Bytes
Number of Reps = 1000
*** Round Trip Avg = 0.000180 uSec
*** Bandwidth = 4239127 474130 Byte/Sec
= 33.013020 Megabit/Sec

Doing round trip test for:
Message length = 10000 integer value(s)
Message size = 40000 Bytes
Number of Reps = 1000
*** Round Trip Avg = 0.009834 uSec
*** Bandwidth = 8094288 734330 Byte/Sec
= 64.754310 Megabit/Sec

Doing round trip test for:
Message length = 100000 integer value(s)
Message size = 400000 Bytes
Number of Reps = 1000
*** Round Trip Avg = 0.966779 uSec
*** Bandwidth = 9229509 668330 Byte/Sec
= 73.838557 Megabit/Sec

[ibrahim@master ~/]
```

Figure 3.11 output MPI code
Chapter Four
4.1 The Benchmarks

A benchmark could be thought of as a measurement tool. Considering the benchmark in computing terminology simply it is a program or application that is specially designed to provide measurements for a specific software or application.

Many benchmark programs exist to measure the cluster system such as LINPACK or the NASA developed NPB (NAS Parallel Benchmark) package.

4.2 The NAS Parallel Benchmarks

NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks (NPB's) were derived from computational fluid dynamics (CFD) codes. They were designed to evaluate the performance of parallel computers and are widely recognized as a standard pointer of computer performance. NPB consists of five kernels and three simulated CFD applications derived from important classes of aero physics applications. These five kernels imitate the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. The benchmarks are specified only algorithmically ("pencil and paper" specification) and referred to as NPB 1, in order to have the complete and concise concept refer to the following link <http://www.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf>[accessed 11.12.2011]. The NPB suite contains eight benchmarks comprising five
kernels (CG, FT, EP, MG, and IS) and three compact applications (BT, LU, and SP).

4.2.1 The Embarrassingly Parallel (EP) Benchmark

The first of the five kernel benchmarks is an Embarrassingly Parallel benchmark. It generates pairs of Gaussian random deviates according to a specific scheme. The goal is to establish the reference point for peak performance of a given platform. EP is almost independent of the interconnectivity of the cluster.

4.2.2 Multigrid (MG) Benchmark

MG uses a V-cycle MultiGrid method to compute the solution of the 3D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between rude and fine, being able to test both short and long distance data movement.

4.2.3 Conjugate Gradient (CG) Benchmark

CG uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries.

4.2.4 FT Benchmark

FT contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension.
4.2.5 Integer Sort (IS) Benchmark

This benchmark tests a sorting operation that is important in particle method codes. This type of application is similar to particle-in-cell applications of physics, wherein particles are assigned to cells and may drift out. The sorting operation is used to reassign particles to the appropriate cells. This benchmark tests both integer computation speed and communication performance. The uniqueness of the problem here is that the floating point arithmetic is not drawn in. However Significant data communication is required.

4.2.6 Simulated CFD Application Benchmarks

The three simulated CFD application benchmarks are intended to accurately represent the principal computational and data movement requirements of modern CFD applications.

4.2.7 LU Benchmark

LU is a simulated CFD application uses symmetric successive overrelaxation (SSOR) method to solve a seven block diagonal system resulting from finite difference discretization of the NavierStokes equations in 3D by splitting to into block Lower and Upper triangular systems.

4.2.8 BT Benchmark

BT is a simulated CFD application that uses an implicit algorithm to solve 3dimensional (3D) compressible NavierStokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y, and z
dimensions. The resulting systems are BlockTridiagonal of 5x5 blocks and are solved sequentially along each dimension.

4.2.9 SP Benchmark

SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam Warming approximate factorization that decouples the x, y, and z dimensions. The resulting system has scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.

The NPB were designed with some classes of problems making kernels harder to compute by modifying the size of data and/or the number of iterations. There are six classes of problems: S, W, A, B, C and D. Class S is the easiest problem and is for testing purpose only. Class D is the hardest and usually requires a lot of memory.
Chapter Five
5 Results and analysis

5.1 Introduction

A number of test benchmarks were conducted on a cluster of four workstations. The benchmarks are standard and are explained in the previous chapter. Numerous clustering parallelism studies was executed from which the researcher picked the following study upon a Linux cluster compromised of four nodes in table 5.1 summary for each node features. The study was carried out through NPB packets of different size respectively packet of size A and packet of size B, keeping in mind that the applied benchmark NAS NPB3.3:

5.2 Machine Data:

<table>
<thead>
<tr>
<th>Machine Name</th>
<th>Kronos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>2.6.10-1.9_FC2kronos</td>
</tr>
<tr>
<td>CPU 1 Model</td>
<td>AMD Sempron(tm) 2500+</td>
</tr>
<tr>
<td>CPU 1 MHz</td>
<td>1753.796</td>
</tr>
</tbody>
</table>

According to the previous processor type and model the following summary of benchmark results are obtained with respect to packet of size A and B upon four processors<http://clustemonkey.net/download/kronos/bps-logs/> [accessed 2.10.2011].
5.2.1 Reference Results Summary for class A

Table 5.2: summary of reference result class A np 4

<table>
<thead>
<tr>
<th>Test Name</th>
<th>MOPS Total</th>
<th>MOPS/CPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>795.01</td>
<td>198.75</td>
<td>211.68</td>
</tr>
<tr>
<td>CG</td>
<td>319.15</td>
<td>79.79</td>
<td>4.69</td>
</tr>
<tr>
<td>EP</td>
<td>31.47</td>
<td>7.87</td>
<td>17.06</td>
</tr>
<tr>
<td>FT</td>
<td>475.29</td>
<td>118.82</td>
<td>15.02</td>
</tr>
<tr>
<td>IS</td>
<td>13.67</td>
<td>3.42</td>
<td>6.13</td>
</tr>
<tr>
<td>LU</td>
<td>897.72</td>
<td>224.43</td>
<td>132.89</td>
</tr>
<tr>
<td>MG</td>
<td>643.21</td>
<td>160.80</td>
<td>6.05</td>
</tr>
<tr>
<td>SP</td>
<td>558.10</td>
<td>139.52</td>
<td>152.32</td>
</tr>
</tbody>
</table>

Graphical representation for the reference result summary

Figure 5.1: class A Execution Time accordance
Figure 5.2: Class A Total Mega Flops for 4 nps

Figure 5.3: Class A Mega process per CPU
5.3 Our Machine Data:

Whereas in our study we rely on the following characteristics:

Table 5.3: data of cluster node

<table>
<thead>
<tr>
<th>Machine Name</th>
<th>HP 6100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td></td>
</tr>
<tr>
<td>CPU 1 Model</td>
<td>Genuine Intel Pentium® Pentium 4</td>
</tr>
<tr>
<td>CPU 1 MHz.</td>
<td>3000</td>
</tr>
<tr>
<td>Cache</td>
<td>1024 KB</td>
</tr>
<tr>
<td>Local memory</td>
<td>1 GB</td>
</tr>
</tbody>
</table>

5.3.1 Benchmark result to our cluster according to the A packet size

Table 5.4: SP Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>64x 64x 64</td>
</tr>
<tr>
<td>Iterations</td>
<td>400</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>560.39</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled proc</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>151.70</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>37.92</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>
### Table 5.5: LU Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>64x 64x 64</td>
</tr>
<tr>
<td>Iterations</td>
<td>250</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>377.04</td>
</tr>
<tr>
<td>Total process</td>
<td>4</td>
</tr>
<tr>
<td>Compiled process</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>316.41</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>79.10</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

### Table 5.6: IS Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8388608</td>
</tr>
<tr>
<td>Iterations</td>
<td>10</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>21.36</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled process</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>3.93</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>0.98</td>
</tr>
<tr>
<td>Operation type</td>
<td>keys ranked</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

56
### Table 5.7: FT Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>256x 256x 128</td>
</tr>
<tr>
<td>Iterations</td>
<td>6</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>68.77</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>103.78</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>25.94</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

### Table 5.8: EP Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>536870912</td>
</tr>
<tr>
<td>Iterations</td>
<td>0</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>42.14</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>12.74</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>3.18</td>
</tr>
<tr>
<td>Operation type</td>
<td>Random numbers generated</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

57
### Table 5.9 CG Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>14000</td>
</tr>
<tr>
<td>Iterations</td>
<td>15</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>6.70</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>223.19</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>55.80</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

### Table 5.10 BT Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>64x 64x 64</td>
</tr>
<tr>
<td>Iterations</td>
<td>200</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>456.51</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>368.64</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>92.16</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>
Table 5.11 MG Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>256x 256x 256</td>
</tr>
<tr>
<td>Iterations</td>
<td>4</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>14.12</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled proc</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>275.70</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>68.92</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

5.3.2 The summarized results

Table 5.12: result summary of cluster

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time in seconds</th>
<th>Mop/s total</th>
<th>Mop/s/process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bt</td>
<td>456.51</td>
<td>368.64</td>
<td>92.16</td>
</tr>
<tr>
<td>Cg</td>
<td>6.70</td>
<td>223.19</td>
<td>55.80</td>
</tr>
<tr>
<td>Hp</td>
<td>42.14</td>
<td>12.74</td>
<td>3.18</td>
</tr>
<tr>
<td>Ft</td>
<td>68.77</td>
<td>103.78</td>
<td>25.94</td>
</tr>
<tr>
<td>ls</td>
<td>21.36</td>
<td>3.93</td>
<td>0.98</td>
</tr>
<tr>
<td>Lu</td>
<td>377.04</td>
<td>316.74</td>
<td>79.10</td>
</tr>
<tr>
<td>Sp</td>
<td>560.39</td>
<td>151.70</td>
<td>37.92</td>
</tr>
<tr>
<td>Mg</td>
<td>14.12</td>
<td>275.70</td>
<td>68.92</td>
</tr>
</tbody>
</table>
5.3.3 Graphically representation of the result

Figure 5.4: class A Benchmark Execution Time

Figure 5.5: Class A Benchmark for Total Mega Flops
Figure 5.6: Class A Benchmark for Mega process per second.

5.4 Reference Summary Results for class B

Table 5.13: summary of reference result class B np 4

<table>
<thead>
<tr>
<th>Test Name</th>
<th>MOPS Total</th>
<th>MOPS/CPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>210.78</td>
<td>52.69</td>
<td>259.56</td>
</tr>
<tr>
<td>EP</td>
<td>31.41</td>
<td>7.85</td>
<td>68.36</td>
</tr>
<tr>
<td>IS</td>
<td>25.29</td>
<td>6.32</td>
<td>13.27</td>
</tr>
<tr>
<td>LU</td>
<td>892.16</td>
<td>223.04</td>
<td>559.12</td>
</tr>
<tr>
<td>MG</td>
<td>679.55</td>
<td>169.89</td>
<td>28.64</td>
</tr>
<tr>
<td>SP</td>
<td>618.15</td>
<td>154.54</td>
<td>574.31</td>
</tr>
</tbody>
</table>
5.4.1 Graphical representation of reference result Class B

Figure 5.7: Reference class B Benchmark result for Execution Time

Figure 5.8: Reference Class B Benchmark result for Total of Mega Flops
Figure 5.9: Reference class B Benchmark result for Process per CPU

5.5 Class B Benchmark results in accordance to our architecture

Table 5.14: SP Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>102x 102x 102</td>
</tr>
<tr>
<td>Iterations</td>
<td>400</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>2829.17</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled process</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>125.48</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>31.37</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>
Table 5.15: MG Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>256x 256x 256</td>
</tr>
<tr>
<td>Iterations</td>
<td>20</td>
</tr>
<tr>
<td>Time in</td>
<td>64.18</td>
</tr>
<tr>
<td>seconds</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>4</td>
</tr>
<tr>
<td>processes</td>
<td></td>
</tr>
<tr>
<td>Compiled</td>
<td>4</td>
</tr>
<tr>
<td>procs</td>
<td></td>
</tr>
<tr>
<td>Mop/s total</td>
<td>303.22</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>75.81</td>
</tr>
<tr>
<td>Operation</td>
<td>floating point</td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile</td>
<td>10 Jan 2012</td>
</tr>
<tr>
<td>date</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.16: LU Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>102x 102x 102</td>
</tr>
<tr>
<td>Iterations</td>
<td>250</td>
</tr>
<tr>
<td>Time in</td>
<td>1902.41</td>
</tr>
<tr>
<td>seconds</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>4</td>
</tr>
<tr>
<td>processes</td>
<td></td>
</tr>
<tr>
<td>Compiled</td>
<td>4</td>
</tr>
<tr>
<td>procs</td>
<td></td>
</tr>
<tr>
<td>Mop/s total</td>
<td>262.21</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>65.55</td>
</tr>
<tr>
<td>Operation</td>
<td>floating point</td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile</td>
<td>10 Jan 2012</td>
</tr>
<tr>
<td>date</td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.17: IS Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>33554432</td>
</tr>
<tr>
<td>Iterations</td>
<td>10</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>73.43</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>4.57</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>1.14</td>
</tr>
<tr>
<td>Operation type</td>
<td>Keys ranked</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

### Table 5.18: FT Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>512x 256x 256</td>
</tr>
<tr>
<td>Iterations</td>
<td>20</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>884.50</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>104.07</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>26.02</td>
</tr>
<tr>
<td>Operation type</td>
<td>Floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

65
Table 5.19: EP Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>2147483648</td>
</tr>
<tr>
<td>Iterations</td>
<td>0</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>169.98</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>12.63</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>3.16</td>
</tr>
<tr>
<td>Operation type</td>
<td>Random numbers generated</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

Table 5.20: CG Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>75000</td>
</tr>
<tr>
<td>Iterations</td>
<td>75</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>291.50</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>187.68</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>46.92</td>
</tr>
<tr>
<td>Operation type</td>
<td>Floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>
Table 5.21: BT Benchmark result

<table>
<thead>
<tr>
<th>Class</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>102x 102x 102</td>
</tr>
<tr>
<td>Iterations</td>
<td>200</td>
</tr>
<tr>
<td>Time in seconds</td>
<td>1905.16</td>
</tr>
<tr>
<td>Total processes</td>
<td>4</td>
</tr>
<tr>
<td>Compiled procs</td>
<td>4</td>
</tr>
<tr>
<td>Mop/s total</td>
<td>368.57</td>
</tr>
<tr>
<td>Mop/s/process</td>
<td>92.14</td>
</tr>
<tr>
<td>Operation type</td>
<td>floating point</td>
</tr>
<tr>
<td>Verification</td>
<td>SUCCESSFUL</td>
</tr>
<tr>
<td>Version</td>
<td>3.3</td>
</tr>
<tr>
<td>Compile date</td>
<td>10 Jan 2012</td>
</tr>
</tbody>
</table>

5.6 The summarized results

Table 5.22: Summarized result

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time in seconds</th>
<th>Mop/s total</th>
<th>Mop/s/cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bt</td>
<td>1905.16</td>
<td>368.57</td>
<td>92.14</td>
</tr>
<tr>
<td>Cg</td>
<td>291.5</td>
<td>187.68</td>
<td>46.92</td>
</tr>
<tr>
<td>Ip</td>
<td>169.98</td>
<td>12.63</td>
<td>3.16</td>
</tr>
<tr>
<td>Ii</td>
<td>884.5</td>
<td>104.07</td>
<td>26.02</td>
</tr>
<tr>
<td>Is</td>
<td>73.43</td>
<td>4.57</td>
<td>1.14</td>
</tr>
<tr>
<td>Lu</td>
<td>1902.41</td>
<td>262.21</td>
<td>65.55</td>
</tr>
<tr>
<td>Mg</td>
<td>64.18</td>
<td>303.22</td>
<td>75.81</td>
</tr>
<tr>
<td>Sp</td>
<td>2829.17</td>
<td>125.48</td>
<td>31.37</td>
</tr>
</tbody>
</table>
5.6.1 Graphical representation of Class B Benchmark results:

Figure 5.10: Class B Benchmark Execution Time

Figure 5.11: Class B Benchmark for Total Mega Flops
5.7 The Comparison between study and reference

5.7.1 Class A

Figure 5.13: The Comparison between study and reference Execution time
Comparison Between Study and Referenced Total Mega Flops

Figure 5.14: The Comparison between study and Reference total Mega flops

Comparison Between Study and Referenced Processes per CPU

Figure 5.15: The Comparison between study and Reference Processes per CPU
5.7.2 Class B

Figure 5.16: The Comparison between study and Reference Time

Figure 5.17: The Comparison between study and Reference total Mega Flops
5.8 Summary of the comparisons

The summary of conducted comparative study according to previous studies and our cluster under the same conditions except for the architecture of the nodes in terms of a number of parameters is tabulated consequently:
5.8.1 For class A packet size

Table 5.23:A. Execution Time analysis

<table>
<thead>
<tr>
<th>Packet Class/Benchmark Test Package</th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A Reference Study Results</td>
<td>Appropriate Throughput</td>
<td>Minimum execution time through all SW packages</td>
<td>Adequate Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class A Research Study Results</td>
<td>Double Time value</td>
<td>Exceeding 2 Seconds</td>
<td>Triple Time value</td>
<td>Quadraple Time</td>
<td>Triple Time value</td>
<td>Quadruple Time</td>
<td>Double Time value</td>
<td></td>
</tr>
<tr>
<td>Packet Class/Benchmark Test Package</td>
<td>BT</td>
<td>CG</td>
<td>EP</td>
<td>FT</td>
<td>IS</td>
<td>LU</td>
<td>SP</td>
<td>MG</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------</td>
<td>----</td>
<td>-----------------------------------</td>
<td>----</td>
<td>--------------------------</td>
<td>--------------------------------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>Class A Reference Study Results</td>
<td>Double Throughput</td>
<td></td>
<td>Magnificent output</td>
<td></td>
<td>Minimum total processes</td>
<td>Maximum total processes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class A Research Study Results</td>
<td></td>
<td></td>
<td>One and half timesage decrease in Throughput</td>
<td></td>
<td>Four times lesser than reference study</td>
<td>Two and half lesser than reference study</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Four times lesser than reference study</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Approximately three times lesser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.25: A Megaflop per processes

<table>
<thead>
<tr>
<th>Packet Class/Benchmark Test Package</th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A Reference Study Results</td>
<td></td>
<td></td>
<td></td>
<td>Almost each CPU executed five times no of processes with respect to the research study</td>
<td>Minimum no of processes</td>
<td>Maximum Throughput which is 3 times better in throughput</td>
<td>Quadruple increase in processes per CPU</td>
<td></td>
</tr>
<tr>
<td>Class A Research Study Results</td>
<td></td>
<td></td>
<td></td>
<td>Maximum value occurred here</td>
<td>Almost half of each CPU processes are executed</td>
<td>Half No processes executed</td>
<td>Minimal no of process for CPU</td>
<td>Second maximum value</td>
</tr>
</tbody>
</table>
Summary

Class A results comparison indicate a stable and equal time stage of decreasing variation towards referenced value through different benchmarking packages used. This is probably due to the difference between the referenced nodes architecture and our study node architecture. Hence we conclude that in order to enhance throughput and efficiency, it is fully recommended to choose and apply the referenced study node architecture in constructing a cluster.
### 5.8.2 For Class B packet size

Table 5.26: Execution Time for class B

<table>
<thead>
<tr>
<th>Packet Class/Benchmark Test Package</th>
<th>CG</th>
<th>EP</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class B Reference Study Results</td>
<td>No significant difference occur</td>
<td></td>
<td>Minimal execution time</td>
<td>Faster in execution with 3.2 times approx.</td>
<td>Maximum; Faster in execution with 5 times approx.</td>
<td></td>
</tr>
<tr>
<td>Class B Research Study Results</td>
<td></td>
<td>More than 2.5 latency times approx.</td>
<td>Five times latency time sages approx.</td>
<td></td>
<td></td>
<td>3 times slower in execution.</td>
</tr>
</tbody>
</table>
Table 5.27: Total Megaflop for class B

<table>
<thead>
<tr>
<th>Packet Class/Benchmark Test Package</th>
<th>CG</th>
<th>EP</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class B Reference Study Results</td>
<td></td>
<td></td>
<td>Minimum; More than five times double throughput</td>
<td>Maximum;</td>
<td>Five times doubled throughput</td>
<td></td>
</tr>
<tr>
<td>Class B Research Study Results</td>
<td>No significant variation occurs</td>
<td>More than double throughput</td>
<td>Minimum;</td>
<td>Approximately 3.5 times degrading throughput</td>
<td>Maximum;</td>
<td>Approximately doubled throughput</td>
</tr>
</tbody>
</table>
Table 5.28: Megaflop per processor for class B

<table>
<thead>
<tr>
<th>Class/Benchmark Test Package</th>
<th>CG</th>
<th>EP</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class B Reference Study Results</td>
<td>Almost equal to no of each CPU processes executed</td>
<td>Almost half of each CPU processes executed</td>
<td>Minimum; which is one fifth no of processes per CPU</td>
<td>Maximum; Quadruple no of processes is being executed in each CPU</td>
<td>Approximately five times greater in no of processes executed</td>
<td></td>
</tr>
<tr>
<td>Class B Research Study Results</td>
<td></td>
<td></td>
<td>Minimum; no of processes per CPU</td>
<td></td>
<td></td>
<td>Maximum; More than 2.5 times lesser in processes per CPU</td>
</tr>
</tbody>
</table>
Summary

From the above tables we argue that comparing packet class A with packet class B still the former conclusion and recommendation stands valid, although for packet class B no significant variation occurs through CG benchmark SW package in between the referenced and the research study experiment.
5.9 Latency and Bandwidth

Latency refers to the time it takes for a single packet to leave the source and reach the destination. It is measured from the time the sending process sends the packet to the time destination process starts receiving the packet. And it is measured by sending a small message back and forth. If the packet was able to make $N$ round-trips in time $T$, the one-way latency of the network will be $T/2N$.

Bandwidth refers to the amount of data that can flow through the interconnect structure in a unit of time.

And we use OSU Micro-Benchmarks 3.5.1 which developed by Ohio State University to test our cluster latency and Bandwidth. The latency tests were carried out in a ping-pong fashion. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test were carried out and average one-way latency numbers were obtained. Blocking version of MPI functions (MPI_Send and MPI_Recv) were used in the tests. <http://mvapich.cse.ohio-state.edu/benchmarks/>[24.12.2011]. And we get the flowing result
### 5.9.1 Latency and bandwidth result

Table 5.29: Result of OSU benchmark

<table>
<thead>
<tr>
<th>Size in kb</th>
<th>Latency (μs)</th>
<th>MB/s</th>
<th>Messages/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48.95</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>50.27</td>
<td>0.01</td>
<td>6924.99</td>
</tr>
<tr>
<td>2</td>
<td>50.27</td>
<td>0.15</td>
<td>72737.16</td>
</tr>
<tr>
<td>4</td>
<td>50.4</td>
<td>0.28</td>
<td>71122.22</td>
</tr>
<tr>
<td>8</td>
<td>50.49</td>
<td>0.56</td>
<td>69663.26</td>
</tr>
<tr>
<td>16</td>
<td>50.98</td>
<td>0.17</td>
<td>10815.29</td>
</tr>
<tr>
<td>32</td>
<td>52.75</td>
<td>0.66</td>
<td>20674.60</td>
</tr>
<tr>
<td>64</td>
<td>55.82</td>
<td>1.16</td>
<td>18164.74</td>
</tr>
<tr>
<td>128</td>
<td>62.62</td>
<td>3.67</td>
<td>28685.27</td>
</tr>
<tr>
<td>256</td>
<td>73.72</td>
<td>5.85</td>
<td>22835.86</td>
</tr>
<tr>
<td>512</td>
<td>103.45</td>
<td>7.07</td>
<td>13814.42</td>
</tr>
<tr>
<td>1024</td>
<td>184.48</td>
<td>9.00</td>
<td>8792.41</td>
</tr>
<tr>
<td>2048</td>
<td>409.67</td>
<td>9.04</td>
<td>4413.04</td>
</tr>
<tr>
<td>4096</td>
<td>752.81</td>
<td>9.78</td>
<td>2387.98</td>
</tr>
<tr>
<td>8192</td>
<td>1738.99</td>
<td>9.62</td>
<td>1174.45</td>
</tr>
<tr>
<td>16384</td>
<td>3324.47</td>
<td>9.87</td>
<td>602.27</td>
</tr>
<tr>
<td>32768</td>
<td>7091.55</td>
<td>9.80</td>
<td>299.12</td>
</tr>
<tr>
<td>65536</td>
<td>13732.15</td>
<td>9.88</td>
<td>150.72</td>
</tr>
<tr>
<td>131072</td>
<td>27494.34</td>
<td>9.79</td>
<td>74.71</td>
</tr>
<tr>
<td>262144</td>
<td>53070.36</td>
<td>9.89</td>
<td>37.74</td>
</tr>
<tr>
<td>524288</td>
<td>105756.94</td>
<td>9.91</td>
<td>18.91</td>
</tr>
<tr>
<td>1048576</td>
<td>212491.3</td>
<td>9.91</td>
<td>9.45</td>
</tr>
<tr>
<td>2097152</td>
<td>424005.99</td>
<td>9.94</td>
<td>4.74</td>
</tr>
<tr>
<td>4194304</td>
<td>844447.34</td>
<td>9.96</td>
<td>2.37</td>
</tr>
</tbody>
</table>

82
5.9.2 Graphical representation of latency and bandwidth

Figure 5.19: latency

Figure 5.20: latency
Figure 5.21: latency scale

From the result and figure we get that latency increasing with the increasing of data size

Figure 5.22: Bandwidth

From the result and figure we get that the bandwidth increasing with increasing of data and be stable near 10 MB/s

84
5.10 Speed scaling of our cluster

We use mpi code of matrix multiply with large number of rows and columns to test the speed of our cluster through running the code in different number of node.

5.10.1 The (MPI) Matrix Multiplication A*B = C code

```c
#include "stdio.h"
#include "mpi.h"
#define N 500 /* number of rows and columns in matrix */

MPI_Status status;

double a[N][N], b[N][N], c[N][N]; //matrix used

main(int argc, char **argv){
    struct timeval start, stop;
    int numberOfTasks,
        mtype,
        taskId,
        numberOfWorkers,
        source,
        destination,
        rows,
        averageRow,
```

85
extra,
offset,i,j,k;
//first initialization
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskID);
MPI_Comm_size(MPI_COMM_WORLD, &numberOfTasks);

numberOfWorkers = numberOfTasks-1;

//-- ------------------------------- master ------------------------------- --

if (taskID == 0) {
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            a[i][j]= 1.0;
            b[i][j]= 2.0;
        }
    }

    /* send matrix data to the worker tasks */

getimeofday(&start, 0);

    averageRow = N/numberOfWorkers; //average rows per worker
    extra= N%numberOfWorkers;   //extra rows
    offset = 0;

    for (destination=1; destination<=numberOfWorkers; destination++){
        if(destination<=extra){

            86
rows = averageRow+1;
}
else{
    rows = averageRow;
}

mtype = 1;
MPI_Send(&offset, 1, MPI_INT, destination, mtype, MPI_COMM_WORLD);
MPI_Send(&rows, 1, MPI_INT, destination, mtype, MPI_COMM_WORLD);
MPI_Send(&a[offset][0], rows*N, MPI_DOUBLE, destination, mtype, MPI_COMM_WORLD);
MPI_Send(&b, N*N, MPI_DOUBLE, destination, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
}

/* wait for results from all worker tasks */
for (i=1; i<=numberOfWorkers; i++){
    mtype = 2;
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*N, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
}

timeofday(&stop, 0);
    printf("Upper Left = %6.2f Upper Right = %6.2f
\n", c[0][0], c[0][N-1]);
    printf("Lower Left = %6.2f Lower Right = %6.2f
\n", c[N-1][0], c[N-1][N-1]);

87
fprintf(stdout,"Time = \n%.6f\n\n", (stop.tv_sec+stop.tv_usec*1e-6)-
(start.tv_sec+start.tv_usec*1e-6));
}

/*-------------------------------- worker ------------------------
 *--------------------------------*/

if (taskID > 0) {
    source = 0;
    mtype = 1;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rows*N, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&b, N*N, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);

    /* Matrix multiplication */
    for (k=0; k<N; k++)
        for (i=0; i<rows; i++) {
            c[i][k] = 0.0;
            for (j=0; j<N; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
        }

    mtype = 2;
    MPI_Send(&offset, 1, MPI_INT, 0, mtype,
MPI_COMM_WORLD);
MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
MPI_Send(&c, rows*N, MPI_DOUBLE, 0, mtype,
MPI_COMM_WORLD);
}

MPI_Finalize();
}

5.10.2 The result after running the code

1. Number of rows and columns 500

[ibrahim@master New Folder]$ mpirun n0-1 ff
Upper Left = 1000.00  Upper Right = 1000.00
Lower Left = 1000.00  Lower Right = 1000.00
Time = 3.865333

[ibrahim@master New Folder]$ mpirun n0-2 ff
Upper Left = 1000.00  Upper Right = 1000.00
Lower Left = 1000.00  Lower Right = 1000.00
Time = 2.379955

[ibrahim@master New Folder]$ mpirun n0-3 ff
Upper Left = 1000.00  Upper Right = 1000.00
Lower Left = 1000.00  Lower Right = 1000.00
Time = 2.025570
2. Number of rows and columns 1000

[ibrahim@master New Folder]$ mpicc -o fff matrix.c
[ibrahim@master New Folder]$ mpirun n0-1 fff
Upper Left = 2000.00   Upper Right = 2000.00
Lower Left = 2000.00   Lower Right = 2000.00
Time = 28.551292

[ibrahim@master New Folder]$ mpirun n0-2 fff
Upper Left = 2000.00   Upper Right = 2000.00
Lower Left = 2000.00   Lower Right = 2000.00
Time = 16.019512

[ibrahim@master New Folder]$ mpirun n0-3 fff
Upper Left = 2000.00   Upper Right = 2000.00
Lower Left = 2000.00   Lower Right = 2000.00
Time = 12.327473

3. Number of rows and columns 2000

[ibrahim@master New Folder]$ mpicc -o ff2 matrix.c
[ibrahim@master New Folder]$ mpirun n0-1 ff2
Upper Left = 4000.00   Upper Right = 4000.00
Lower Left = 4000.00   Lower Right = 4000.00
Time = 226.229442
5.10.3 Summarizing result

Table 5.30: summarizing the MPI code result

<table>
<thead>
<tr>
<th>Number of node</th>
<th>Time 500</th>
<th>Time 1000</th>
<th>Time 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.865333</td>
<td>28.551292</td>
<td>226.229442</td>
</tr>
<tr>
<td>3</td>
<td>2.379955</td>
<td>16.019512</td>
<td>119.541475</td>
</tr>
<tr>
<td>4</td>
<td>2.025570</td>
<td>12.327473</td>
<td>86.513777</td>
</tr>
</tbody>
</table>
5.10.4 Graphical representation

Figure 5.23: number of rows and columns 500

Figure 5.24: number of rows and columns 1000
From the result and figures above we get that the run time was decreasing with increasing of number of processors.
Chapter Six
Conclusion and future work

6.1 Conclusion

Parallel computing is a new and rapidly diverging technology in nowadays computation. Clusters of computers are massively needed for scientific and computation necessities. What the researcher had concluded is that building the best computer clusters needs heterogeneous hardware with high RAM, Processor and networking infrastructure.
Moreover, during performance tests we found that certain applications gave magnificent results for high response time and throughput in such a way that it is incomparable to single systems; accordingly it is best recommended for such applications to utilize parallel systems.
Message passing interface was found to be of significant influence on parallel computing environment and other distributed memory systems. Where it availed different libraries for dealing with parallel processes through a system of distributed processors. Advances in networking technology and distributed systems created great capability for dynamic task management in MPI.
6.2 Recommendation

According to what the researcher had concluded it is worth to depend on clustering systems as parallelism agents but to go further we recommend to:

- It proved efficiency and reliability than ordinary computing environment with few capabilities, so we recommend to rely on such infra on scientific research and extensive computation applications with high floating points and high demand on throughput.
- Extensive training and advertising through conferences, journals and papers and encouraging parallel computing and availing all kind of support for an efficient environment and benefit proof leads to rapid development in this field which narrow the gap between former successors.
- Scientific institutes, Universities and organizations are encouraged and should rely on such economical environment and support researchers and implementers.
- Enhance the parallel cluster environment by:
  1. Providing enhanced features hardware for cluster nodes.
  2. A stable and appropriate environment such as electricity, ventilation and network infra should be availed.
  3. Better inter and outside connectivity must exist for better usage and benefit.
  4. Number of nodes should be increased.
References

Bookman C, 2002. Linux clustering building and maintaining
Publisher: New Riders Publishing

Collings T, 2002. Redhat Linux Networking and system administration
Published by Hungry Minds, Inc. 909 Third Avenue
New York, NY 10022

Published: by Elsevier Science (USA)

El-rwini H, 2005. Advanced computer architecture and parallel processing
Published by John Wiley & Sons, Inc, Hoboken, New Jersey.
Published simultaneously in Canada.

Rauber T, 2010. Parallel programming for multicore and cluster system
Published in the book series: Springer-Lehrbuch

Published in 2006 by Chapman & Hall/CRC


<http://www.computerpoweruser.com/articles/archive/c0604/29c04/29c04.pdf?guid=>
http://www.nas.nasa.gov/

<http://mvapich.cse.ohio-state.edu/>