Empirical Performance Analysis of Parallel Programs on Wired and Wireless Local Area Networks

By

Saidu, Isah Charles

Department of Mathematics

Faculty of Science,

Ahmadu Bello University Zaria, Nigeria

May, 2014

Empirical Performance Analysis of Parallel Programs on Wired and Wireless Local Area Networks

By

Saidu, Isah Charles, B.Tech Computer Science (FUTY) 2008

(MSC/SCIE/10371/2010-2011)

A Thesis submitted to the School of Postgraduate Studies,

Ahmadu Bello University, Zaria

In Partial fulfilment of the requirements for the Award of

M.Sc Degree in Computer Science.

Department of Mathematics,

Faculty of Science,

Ahmadu Bello University, Zaria

May, 2014

DECLARATION

I declare that the work in this thesis entitled “Empirical Performance Analysis of Parallel Programs on Wired and Wireless Local Area Networks” has been performed by me in the Department of Mathematics under the supervision of Dr. A. A. Obiniyi and Professor S. B. Junaidu.

The information derived from the literature has been duly acknowledged in the text and the list of references provided. No part of this thesis was previously presented for another degree or diploma at any university.

Saidu, Isah Charles

_____	_____	_____
Name of Student	Signature	Date

CERTIFICATION

This thesis entitled “Empirical Performance Analysis of Parallel Programs in Wired and Wireless Local Area Networks” meets the regulation governing the award of the degree of M.Sc. Computer Science of Ahmadu Bello University, and is approved for its contribution to knowledge and literary presentation.

_____	_____
Chairman, Supervisory Committee Dr. A. A. Obiniyi	Date
_____	_____
Member, Supervisory Committee Prof. S. B. Junaidu	Date
_____	_____
External Examiner	Date
_____	_____
Head of Department, Mathematics Dr. Babangida Sani	Date
_____	_____
Dean, School of Postgraduate Studies Professor A. A. Joshua	Date

DEDICATION

This thesis is dedicated to Almighty God for His love, mercies, faithfulness, kindness and protection towards me, for giving me the privilege to undertake this program. I also dedicate this thesis to my entire paternal and maternal families the Saidus' and the Tongos', especially my father Mr. Saidu Umoru and most loving mother Dr. M. A. Tongo whose encouragements have been made this dream a reality.

ACKNOWLEDGEMENT

I like to, first and foremost express my sincere gratitude to God Almighty for His everlasting love, kindness, mercies, protection, wisdom and for sustaining me throughout the period of this research work.

I wish to express my sincere appreciation and special thanks to my supervisors Dr. A. A. Obiniyi (Major) and Professor S. B. Junaidu (Minor) for sacrificing their time, patience and commitment in the actualization and completion of this thesis despite their crowded schedules.

My appreciation also goes to the Head and lecturers of Department of Mathematics for their support towards the success of this thesis.

A special acknowledgement goes to my parents the Saidus' and the Tongos with special acknowledgement to my mother Dr. M. A. Tongo for always being a source of inspiration and encouragements.

I also would like to acknowledge Dr. Peter Ogebebe, Head of Department Computer Science, Bingham University, Dr. N. V. Blamah of the Department of Computer Science, University of Jos, Dr. Yahaya of Deparment of Mathematics ABU Zaria and Dr. Babangida Sani for their support in difficult times.

Lastly my gratitude goes to all my colleagues and well-wishers who contributed in one way or the other towards the completion of this thesis, just to mention but a few; Mr. John Ogbiti, Miss. Comfort George, Mrs. Florence Chukwuemaka, Mr. Babatunde Jacob and the students of Computer Science Department, Bingham university, whose laptops form part of the laptops used for the experiments in this work

ABSTRACT

High speed and reduced cost of hardware have always been a key motivating factor for innovative concepts in computer performance. At the moment there is a shift from the way computers are built which involves trying to make a single processor run faster to combining several numbers of processors/processing units to simultaneously solve a task. A special kind of High performance computer known as the Beowulf cluster is the result of this innovation. This thesis focuses on measuring performance of parallel programs deployed on wired and wireless networks. Our experiments were conducted on Beowulf clusters. It has been shown empirically in this work that parallel programs written for Beowulf clusters on Wireless Local Area Network (LAN: IEEE 802.11) do not gain appreciable speedup as the number of processing nodes increases compared to the same parallel programs written for the same Beowulf clusters but on wired LAN. In order to improve performance of parallel programs on wireless networks without exploiting any thread level parallelism, a multiprogramming approach was used to localize communication and reduce network overhead thereby improving performance

TABLE OF CONTENTS **PAGE**

DECLARATION iii
CERTIFICATION iv
ACKNOWLEDGEMENT vi
ABSTRACT vii
CHAPTER ONE 1
INTRODUCTION 1
1.1 Background of Study 1
1.2 Motivations for the Study 3
1.4 Research Objectives 5
1.5 Research Methodology 5
1.6 Organization of Study 6
CHAPTER TWO 7
LITERATURE REVIEW 7
2.1 Introduction 7
2.2 Super/Parallel Computers 8
2.2.1 Supercomputers 8
2.2.2 Centralized Parallel Computers 9
2.2.3 Multicore Parallel Computers 9
2.2.4 Multicomputer parallel Computers 9
2.2.5 Beowulf Clusters 9
2.3 Parallel Computations and Models 10
2.3.1 Parallel Communication 10
2.3.2 Shared Memory Model 11
2.4 Parallel Libraries and Parallel Programming Languages 13
2.4.1 MPI (Message Passing Interface) 14
A. DeinoMPI 15
B. OpenMPI 15
2.4.2 Parallel Virtual Machine (PVM) 16
2.4.3 A-BSP 16
2.4.5 TBB 17
2.4.7 Cilk ++ 18
2.5 Wired and Wireless Computer Networks 19

2.5.1 Wired Networks 19
2.5.2 Logical Link Control Sub –Layer 19
2.5.3 Media Access Control Sub –Layer 20
2.5.4 Wired LAN: Ethernet 20
A. Standard Ethernet 20
C. Gigabit Ethernet 22
2.5.4 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) 22
2.5.5 Wireless Networks 24
2.5.6 Wireless Local Area Networks (WLANs) 25
B. IEEE 802.11a 25
C. IEEE 802.11b 26
D. IEEE 802.11g 26
E. IEEE 802.11n 26
2.5.7 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) 27
2.6 Review of Related Works 28
2.7 Summary 29
CHAPTER THREE 30
PARALLEL MATRIX-VECTOR MULTIPLICATION EXPERIMENT 30
3.1 Introduction 30
3.2 Experimental Setup 30
3.2.1 System Configuration 30
3.2.2 Network Setup 30
3.2.3 Operating System 31
3.2.4 Parallel Library Setup (Using DeinoMPI) 31
3.2.5 Program Compilation 32
3.2.6 Execution Plan 33
3.3 Sequential Matrix – Vector Multiplication Algorithm 33
3.4 Parallel Matrix-Vector Multiplication Algorithm 33
3.4.1 Parallel Matrix-Vector Multiplication Algorithm (Row-wise Decomposition) 34
3.4.2 Execution Plan 37
3.4.3 Results of Parallel Matrix-Vector Multiplication Algorithm (Row-wise Decomposition) on Wired and Wireless LAN 38

3.4.4 Parallel Matrix-Vector Multiplication Algorithm (Column-wise Decomposition) 43
3.4.5 Execution Plan 47
3.4.6 Results of Parallel Matrix-Vector Multiplication Algorithm (Column-wise Decomposition) on Wired and Wireless LAN 48
3.4.7 Parallel Matrix-Vector Multiplication Algorithm (Checkerboard Decomposition) 52
3.4.9 Results of Parallel Matrix-Vector Multiplication Algorithm (Checkerboard Decomposition) on Wired and Wireless LAN 57
3.5 Chapter Summary....62
CHAPTER FOUR....63
PARALLEL MATRIX-MATRIX MULTIPLICATION EXPERIMENT 63
4.1 Introduction 63
4.3 Sequential Matrix-Matrix Multiplication Algorithm 63
4.4 Parallel Matrix-Matrix Multiplication Algorithm (Row-wise decomposition)....64
4.5 Execution Plan....70
4.6 Results Parallel Matrix-Matrix Algorithm (row-wise decomposition) on Wired and Wireless LAN....71
4.7 General Observation of the Algorithms 73
4.8 Multiprogramming approach to performance improvement....81
4.8.1 Multiprogramming: Two Processing Nodes....81
4.8.2 Multiprogramming: Three Processing Nodes:....82
4.8.3 Multiprogramming: Four Processing Nodes....84
4.8.4 Multiprogramming: Five Processing Nodes 86
CHAPTER FIVE 88
SUMMARY AND CONCLUSION 88
5.1 Summary 88
5.2 Conclusions....89
5.3 Recommendation for Further Study....90
REFERENCES....92
APPENDIX A....94
APPENDIX B 99
APPENDIX C 124

LIST OF FIGURES

PAGE

Figure 2.1: Shared memory model……. 11

Figure 2.2: Distributed memory model…….. 12

Figure 3.1: Row-wise decomposition of matrix…… 35

Figure 3.2: Primitive task involving the multiplication of row i of A and vector b(row-wise decomposition)……... 35

Figure 3.3: Performance for matrix-vector multiplication using row-wise decomposition…… 39

Figure 3.4: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition) on 2 cluster wired LAN…….. 41

Figure 3.5: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition) on 3 cluster wired LAN…….. 41

Figure 3.6: Jumpshot view of parallel matrix-vector multiplication (row-wisedecomposition) on 2 cluster wireless LAN……. 42

Figure 3.7: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition) on 3 cluster wireless LAN…….. 42

Figure 3.8: Column-wise decomposition of matrix……... 44

Figure 3.9: Primitive task involved in a parrallel column-wise decomposed matrix-vector multiplicationalgorithm…… 45

Figure 3.10: Performance for parallel matrix-vector multiplication using column-wise decomposition…… 48

Figure 3.11: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 2 Cluster wired LAN…….**.....** 50

Figure 3.12: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 3 Cluster wired LAN…….**.....** 50

Figure 3.13: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 2 Cluster wireless LAN…….. 51

Figure 3.14: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 3 Cluster wireless LAN……... 51

Figure 3.15: Checkerboard decomposition of parallel matrix-vector algorithm……... 52

Figure 3.16: Partitioning of Vector B and Matrix A (Checkerboard decomposition)……... 53

Figure 3.17: Parrallel Matrix-vector multiplication steps (checkerboard decomposition)…… 54

Figure 3.18: Parrallel Matrix-vector multiplication steps (checkerboard decomposition)…… 56

Figure 3.19: Showing parallel checkerboard matrix-vector multiplication algorithm when number of process is NOT a square……... 56

Figure 3.20: Performance graph for parallel matrix-vector multiplication using checkerboarddecomposition……. 58

Figure 3.21: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 2 Cluster wired LAN…….. 60

Figure 3.22: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 3 Cluster wired LAN…….. 60

Figure 3.23: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 2Cluster wireless LAN…….. 61

Figure 3.24: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 3 Cluster wireless LAN…….. 61

Figure 4.1: Showing the primitive task involved in parallel matrix-matrix multiplication with row-wise decomposition…… 65

Figure 4.2: Partial multiplication of Matrix A and B producing a partial Resultant Matrix C….. 66

Figure 4.3: First phase of the ring- exchange of allotted portion of B…….. 67

Figure 4.4: Second phase of the ring-exchange of allotted portion of B…… 67

Figure 4.5: Third phase of the ring-like exchange of the allotted portion of B……. 68

Figure 4.6: Final phase of the ring-like exchange of the allotted portion of B……. 68

Figure 4.7: Performance graph for Parallel Matrix-Matrix Multiplication (row-base decomposition)……... 74

Figure 4.8: Jumpshot view of parallel matrix-matrix multiplication algorithm (row-wise decomposition) on 2 cluster wired LAN……..…… 76

Figure 4.9: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 5 cluster wired LAN……..…….. 76

Figure 4.10: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 8 cluster wired LAN…….…… 77

Figure 4.11: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithmon 2 cluster wireless LAN……. 78

Figure 4.12: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 5 cluster wireless LAN……. 79

Figure 4.13: Jumpshot view of parallel Matrix-Matrix Multiplication (row-wise decomposition) algorithm on 8 cluster wireless LAN…….. 80

Figure 4.14: Performance graph for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 2……..…… 82

Figure 4.15: Performance graph for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 3……..…… 83

Figure 4.16: Performance graph for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 4…… 85

Figure 4.17: Performance graph for multiple processes - parallel matrix-matrix multiplication algorithmon cluster size 5…… 86

LIST OF ABBREVIATIONS

ANSI	American National Standards Institute
ATM	Asynchronous Transfer Mode
BSP	Bulk Synchronous Parallelism
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CTS	Clear to Send
DCF	Distributed Coordination Function
DSSS	Direct-Sequence Spread Spectrum
ETSI	European Telecommunication Standard Institute
FDDI	Fibre Distributed Data Interface
GUI	Graphical User Interface
HCF	Hybid Coordination Function
HomeRFWG	Home Radio Frequency Working Group
IBM	International Business Machines
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standard Organization
LAN	Local Area Networks
LLC	Logical Link Layer
MAC	Media Access Control
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MPICH	Message Passing Interface Chameleon
NIC	Network Interface Card
OFDM	Orthogonal Frequency Division Multiplexing
OpenMPI	Open Message Passing Interface
PAN	Personal Area Networks
PCF	Point Coordination Function
PRAM	Parallel Random Access Macchine
PVM	Parallel Virtual Machine
RTS	Request to Send
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multiprocessor
TBB	Thread Building Block
UTP	Untwisted Pair Cable
WAN	Wide Area Networks
WECA	Wiress Ethernet Compatibility Alliance
WiMAX	Worldwide Interoperability for Microwave Access

CHAPTER ONE

INTRODUCTION

1.1 Background of Study

Speed is a major driving force when it comes to innovative computer designs. Computers that were regarded as superfast decades ago cannot compete with low-end personal computers of today. The focus has changed from making single processors faster to combining processing power of multiple processors to solve a computational problem.Concepts like parallel computing and multicore-computers are now gaining grounds in the technology world(Pawel and Michal, 2006).

Parallel Computer is a multiple-processor computer system supporting parallel programming (Ananth *et al* 2003). A parallel computer can also be described as Multicomputer Parallel System which is constructed out of multiple computers and an interconnection network (Michael, 2003). The concept of multi-processors took a different dimension with the advent of multi-core computers. (Alfredo *et al* 2012) described Multi-core computers as computers whose architecture follow the Thread Level Parallelism(TLP) where higher performance can be achieved by replicating execution units (or cores) on the die while keeping the clock rates in a range where power consumption and heat dissipation do not represent a problem.

This thesis focuses on measuringperformance of parallel programs deployed on wired and wireless networks. This particular category of parallel computers are built from everyday computers as opposed to the expensive high-end network of servers. Gropp *et al* (2003)describes these kind of parallel computers as Beowulf Clusters**.**

With proliferation of mobile devices, introduction of multicore/multiprocessor technologies, parallel multicomputer concepts and the advancement in wireless technology, it can be inferred that in the near future a lot of software applications will be deployed on wireless devices and there is a possibility of parallel computations among these wirelessly deployed devices. Since not all computing devices can be connected via wired means of communication due to obvious reasons like distance, cost and ease of deployment, many computing devices have opted to wireless communication even though the bandwidth for wireless networks is significantly less compared to wired networks.

Also networks (wired and wireless) often times experience lags in their performances, therefore the actual throughput for these networks is often time difficult to calculate. This is because in wireless networks, the available bandwidth undergoes fast time-scale variations due to channel fading and error from physical obstacles and in addition the wireless channel is also a shared-access medium, and the available bandwidth varies with the number of hosts contending for the channel(Samarth*et al* 2013).

This can be seen in actual performance of Ethernet LAN and Wireless LAN with capacity of 100mbps and 54mbps (100Base-TX Ethernet and Wireless 802.11), therefore parallel computations on these two types of networks have the tendencies to be affected by these lags.Often times, parallel computations have been measured in terms of speedup with respect to problem size, number of processors and scalability of the computation with respect to the parallel portion of the program while little consideration or concentration is given to the speedup of the computation with respect to network performances, especially when the network experiences lags.

Even though it may be shown mathematically, that parallel programs deployed on wired LAN have the potential of outperforming those deployed on wireless networks, it has not been shown empirically the extent to which the impact of these networks is to the overall performance of parallel programs. The speed of a parallel program running on a multi-computer parallel system is not only determined by the speed of the individual computer, its local memory but also the speed of the network, especially when the parallel algorithm involves a lot of internetwork communication during computation. Hence, for example, if the speed of a parallel program running on an Ethernet LAN with bandwidth of 100mbps is 5s, can we then infer that the speed of the same program running on a wireless network with maximum bandwidth of 54mbps is about two times that of its wired counterpart and can we expect a speedup in performance when we increase the number of participating systems in the wireless network?

In general what possible empirical factors can contribute to the performance improvement/degradation of parallel programs running on wired and wireless networks?

1.2 Motivations for the Study

The motivation for this research stems from the need to deploy parallel computers remotely. If it is possible for Beowulf clusters to be deployed on wireless networks then this will further reduce to the cost of having a super-computer. Also the issue of mobility of the parallel-computer will no longer be an issue since a node in the cluster can be repositioned without total reconfiguration of the system. This study also serves as an input in the area of parallel computing in mobile devices.

In general this study is aimed at carrying out an empirical study of the performance of parallel programs deployed on wireless system and parallel programs deployed on wired system, with a view to finding out if there are programs that are particularly suited for the wireless environment. The findings of this research will help future design approach to parallel programs deployed across any of these networks.

1.3 Research Questions

The speed of data transfer in wired networks is obviously faster than in wireless networks, but an experimental quantification of the impact of these two types of networks on parallel computation is important in parallel algorithm design. Therefore this research tries to answer the following questions

a. Given a problem ofsize n, network speed s and number of nodes$p$ (computers in the cluster), how does the network speed affect the overall speedup of the parallel program. That is what is the maximum speedup that can be achieved as the speed of the network increases with respect to the problem size n or with respect to the number of nodes p
b. From known facts about wired and wireless LAN, with an example of 100Base-TX Ethernet network and Wireless IEEE 802.11 networks, it can be estimated that the maximum ratio of their bandwidth is at most $100:54$ respectively. Is there a similar performance ratio in terms of speed of computation between parallel programs written for wired and wireless networks?
c. What factors contribute to the observed result and can there be a modification to the parallel program for wireless network to improve its performance? In order words, what factors determine performance differences between parallel programs on wired and wireless networks?

1.4 Research Objectives

a. One of the objectives of this research is to compare performance of parallel programs deployed on wired networks with parallel programs deployed on wireless networks. Hence, this research determines if there is appreciable speedup for parallel programs on wireless networks as the number of computational nodes in the wireless networks increase.

b. Another objective is to determine if there are specific parallel programs that run better when deployed on wireless networks.

1.5 Research Methodology

In this research, a network of ten laptops each running MPI was setup. The wired local area networkwassetup with a switch as the centre device thereby forming a star topology network. The wireless network contains an access point and all laptops are interconnected through the access point.This connectivity also forms a Star topology network.

All the laptops in the cluster run DeinoMPI implementation for Message Passing Interface and each laptop was installed with Windows 7 Operating System..

Four parallel algorithms wereused as case study in this thesis. These algorithms are, Parallel Matrix-Vector (Row-wise Matrix decomposition) Multiplication Algorithm, Parallel Matrix-Vector Multiplication Algorithm (Column-wise decomposition), Parallel Matrix-Vector (Checkerboard decomposition) Multiplication algorithm and Parallel Matrix-Matrix Multiplication Algorithm (Row-wise Decomposition). These algorithms were selected because of the importance of matrix multiplication algorithm in different scientific and commercial applications.

1.6 Organization of Study

The rest of the thesis is organized as follows; chapter two contains literature review of wired and wireless networks, parallel computation in distributed memory parallel system (multicomputer parallel systems), various MPI implementations and a brief history of shared memory parallel programs, Chapter three discussesexperimental setups for parallel matrix-vector multiplication algorithm row-wise decomposition, parallel matrix-vector multiplication algorithm column-wise decomposition and parallel matrix-vector multiplication algorithm checkerboard decomposition.For the comparison, it shows the detailed design principles of these algorithm and implementation processes followed by the results obtained from the experiments. Chapter four discusses parallel matrix-matrix multiplication algorithm row-wise decomposition. It contains detailed experiment setup of the algorithm and details of results obtained from the experiments. Chapter five is for summary, conclusion and future research work.

1.7 Contribution to Knowledge

The findings from this research provides a springboard for further researches into the development of parallel computers built from everyday personal computers and helps to determine the viability of deploying these parallel computers on wireless networks.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

The need for computational speedup is an ever growing problem. Therefore, it becomes worthwhile to look for a different approach to achieving faster computational speed(John and David, 2007). The need for computational speed-up has prompted computer scientists and engineers to employ various means to develop/build faster computers. These approaches are

a. Develop better algorithms for a problem set
b. Redesign machines by introducing the concept of parallel processors where two or more processors run concurrently to solve a computational problem
c. Put together a network of low-end everyday workstations with an additional middleware library to collectively solve a computation problem

The process of putting machines/processors together to collectively solve a program by scheduling portions of the program to run concurrently on each machine/process is the concept known as parallel computing.

It is obvious that the cheapest and possibly the easiest way to achieve speedup is through the process of setting up a network of low-end workstations as noted above. Furthermore, the cost of setting up network infrastructure and the scalability of the network brings to the fore the problem of deciding what type of network(wired or wireless) will be setup to achieve this high-speed computation.To this end, this research focuses on measuring performance of parallel programs on these networks to ascertain the viability of setup of a wireless Beowulf cluster.

Therefore, this research borders around four broad areas categorized under, Super/parallel computers, Parallel Computations and Models, Parallel Programming Languages and Tools, and Wired and Wireless Computer Networks. These areas are reviewed in detail, in this chapter to identify the foundation required for this research.

2.2 Super/Parallel Computers

A parallel computer is a computer that can explicitly process jobs simultaneously.In order words a computer that has the ability to process instructions simultaneously is known as a parallel computer. This concept of simultaneously processing instruction is known as parallel processing. However, the ability of a computer to process jobs simultaneous relies on the architecture of the computer.

With the quest for high speed computers, many categories of parallel computers now exist. These parallel computers can be classified under centralized multiprocessor parallel computers, multi-core processor parallel computers and multicomputer parallel computers with a special category known as Beowulf which is the centre of this research work.

2.2.1 Supercomputers

A supercomputer is a kind of computer that is very powerful in terms of computational capabilities and with multiple high clock-rate processors connected by local high speed buses. These computers are typically very expensive.

According to (Cameron and Tracey, 2003), a supercomputer is the fastest computer of its time. The first supercomputer of history was the Cray-1 which was widely used in the 1980s for large scale simulation.

Today, all supercomputers are parallel computers. Some are based on specialized processors and networks, but the majority are based on commodity hardware and open source operating system and applications software.

2.2.2 Centralized Parallel Computers

A centralized multiprocessor computer also called symmetric multiprocessor or (SMP) is more highly integrated system in which all CPUs share access to a single global memory. This shared memory supports communication and synchronization among processors (Michael, 2003).

2.2.3 Multicore Parallel Computers

A multicore parallel computer is a parallel computer whose computing component has two or more processing units usually called the “core”.

2.2.4 Multicomputer parallel Computers

This is a parallel computer constructed out of multiple computers whichareinterconnected via network. These computers interact by passing information to and fro each other using the network. These networks could be high speed bus on or internetworked computers.

2.2.5 Beowulf Clusters

Beowulf parallel computers were driven by popularity of personal computers and their increasing speed. Beowulf parallel computers are made from off-the-shelf components. Unlike commercial systems, commodity clusters typically are not balanced between compute speed and communication speed; the communication speed is very slow compared to the high speed of processors (Michael, 2003).

Gropp, *et al* (2003), also described a parallel computer system known as Beowulf Cluster which is similar to the concept of Multicomputer Parallel System. According to him, a Beowulf cluster is a form of parallel computer, which is nothing more than a computer that uses more than one processor. There are many different kinds of parallel computers, distinguished by the kinds of processors they use and the way in which those processors exchange data. A Beowulf cluster takes advantage of two commodity components: fast CPUs designed primarily for the personal computer market and networks designed to connect personal computers together (in what is called a local area network or LAN).

Behrooz (2002) described two categories of parallel computers namely Data flow Parallel Computers and Control Flow Parallel Computers. Control-flow parallel computers are essentially based on the same principles as thesequential or von Neumann computer, except that multiple instructions can be executed atany given time. Data-flow parallel computers sometimes referred to as "non-von Neumann,"are completely different in that they have no pointer to active instruction(s) or a locus ofcontrol. The control is totally distributed, with the availability of operands triggering theactivation of instructions.

2.3 Parallel Computations and Models

2.3.1 Parallel Communication

The main key to parallel computations is communication among the processors. There are two general communication models for parallel computations, Shared Memory and Message Passing Models. In this work, these two models are considered

2.3.2 Shared Memory Model

In the shared memory model, each process shares the same memory and therefore thesame data. In this model communication is implicit. The symmetrical multiprocessor or multicore processor makes use of this model. Programing under this model requires care as problems like race conditions, deadlocks and data synchronization can lead to wrong computation since processors share the same memory.

There are a number of differentsub models, such as Parallel Random Access Machine (PRAM) whosesimplicity to understand has led to its popularity. Figure 2.1 illustrates how aPRAM would look, with each processor sharing the same memory and by extensionthe program to execute. However, a pure PRAM machine is impossible to create inreality with a large number of processors due to hardware constraints, so variationsto this model are required in practice(Brown, 2010).

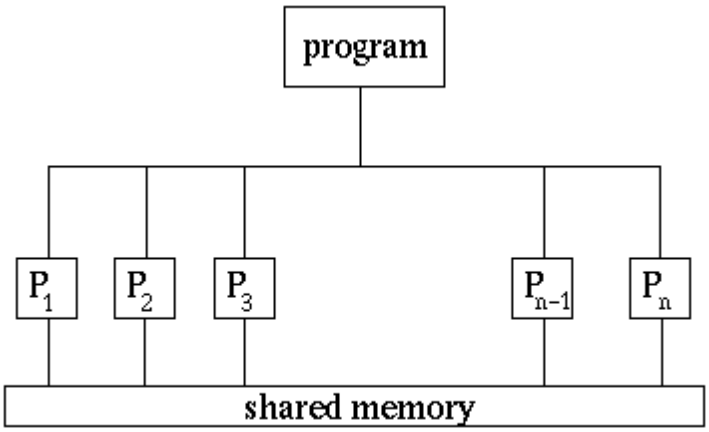


Figure 2.1: Shared memory model(Brown, 2010)

Brown, (2010)reported that “Bulk Synchronous Parallelism (BSP) is a parallel programming model that abstracts from low-level program structures in favour of

supersteps. A superstep consists of a set of independent local computations, followed by global communication phase and barrier synchronization.One of the major advantages of BSP is that the fact that the runtime can easily be deduced. The benefit of BSP is that it imposes a clearly structured communication model upon the programmer.However,extra work is requiredto perform the more complex operations, such as scattering of data.

2.3.3 Message Passing Model

Message Passing Model is a parallel process paradigm widely used by parallel systems with distributed memory.

Unlike the shared memory model, in message passingcommunication is explicit. Figure 2.2 illustrates a typical message passing parallelsystem setup, with each processor equipped with its own services such as memoryand IO. Additionally, each processor has a separate copy of the program to execute(Brown, 2010). One of the major benefits of message passing model is that processors can be added and removed without altering the algorithm design.

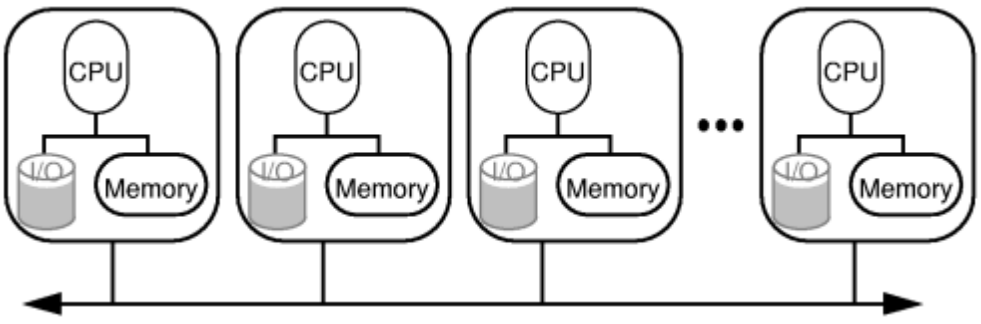


Figure 2.2: Distributed memory model(Brown, 2010)

2.3.4 Computation

Flynn (1972) proposed a four-way classification based their architecture and the way they process information. This classification was later known as the Flynn's Taxonomy. The classification was based on the notion of instruction streams and data streams. These categories are SISD, SIMD, MISD and MIMD.

SISDSingle Instruction Single data (SISD) represents ordinary "uniprocessors" machines. Computers in the SIMDSingle Instruction Multiple Data (SIMD)class, with several processors directed by instructions issued from a central control unit, are sometimes characterized as "Array Processors." Machines in the Multiple Instruction Multiple Data (MISD) category have not found widespread application, but one can view them as generalizedpipelines in which each stage performs a relatively complex operation (as opposed to ordinary pipelines found in modern processors where each stage does a very simple instruction-level operation)(Behrooz, 2002). The Multiple Instruction Multiple Data (MIMD) category includes a wide class of computers. For this reason, in 1988, E. EJohnson proposed a further classification of such machines based on their memory structure(global or distributed) and the mechanism used for communication/synchronization (sharedvariables or message passing)..

2.4 Parallel Libraries and Parallel Programming Languages

Parallel programming is programming in a language that allows you to explicitly indicate how different portions of the computations may be executed concurrently by different processors (Michael, 2003). How to explicitly determine the parallel portion of a program strongly depends on the programming languages.

There are various ways to build languages for parallel computers; these languages can be built by extending an existing compiler, extending a sequential

programminglanguage, adding a parallel programming layer or creating a parallel language from ground-up.In the following sections, detailed information of parallel programming languages and parallel libraries will be discussed.

2.4.1 MPI (Message Passing Interface)

MPI is not a programming language but a standard for message passing. The MPI standard was proposed by a broadly based committee of vendors, implementers, and users. This standard was for high performance on both massively parallel machines and on workstation clusters (MPI, 2013). MPI was designed by a broad group of parallel computer users, vendors, and software writers. These include

a. Vendors - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
b. Library writers - PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
c. Application specialists and consultants

As stated in (MPI, 2013), the first version (MPI Version 1.0) was released in 1994. Subsequent modifications were made to the standard resulting to versions 1.1, 1.2 and the current version 2.0.

MPI is a standard hence there are several implementations in different programming languages. The most widely known implementation is MPICH. MPICH is an open source implementation of MPI and it has been tested on Linux (on IA32 and x86-64), Mac OS/X (PowerPc and Intel), Solaris (32 and 64bit) and Windows. Current version for MPICH is MPICH version 3.0.

Other implementations of the MPI standard have been based on the MPICH implementation. Implementations like MP-MPICH which is a standard for Multi-platform is based on MPICH 1. WINMPICH is an implementation of MPI for

Windows from the Mississippi State University, Cray MPI product for the T3D from Cray Research and The Edinburgh Parallel Computer Centre, HP MPI Implementation for the entire HP product line, IBM's MPI for the SP, DeinoMPI and a host of other MPI implementations. DeinoMPI is an implementation of MPI-2 for Microsoft Windows environment and as such has been used for this research.

A. DeinoMPI

DeinoMPI is an implementation of MPI-2 for Microsoft Windows provided by Deino Software © 2009. The current release version is 2.0.1 and it comes in both Win32 and Win64 machinesDeinoMPI(2013). As state in deinompi (2013), it is a system level middle-ware for high performance parallel computing on Microsoft Windows systems. It provides the libraries necessary to write parallel applications using sequential programming languages like C, C++ and Fortran.

Since Microsoft Windows does not provide native capability to start user applications on remote machines, the process manager provided by deinoMPI makes it possible for processes on multiple machines in a cluster to be securely started remotely(DeinoMPI, 2009). Also shipped with deinoMPI library is the mpiexe.exe, DeinoMPIwin.exe and the Jumpshot. The mpiexec.exe is the MPI job launcher, DeinoMPIwin.exe is a GUI interface which includes mpiexec job launching functionality and setup and management tools while the Jumpshot is a java tool to view log files created when MPI jobs are profiled.

B. OpenMPI

Open MPI is a Message Passing Interface (MPI) library project combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI). It is used by many TOP500 supercomputers including Roadrunner,

which was the world's fastest supercomputer from June 2008 to November 2009, and K computer, the fastest supercomputer from June 2011 to June 2012.

2.4.2 Parallel Virtual Machine (PVM)

PVM is also a popular message passing model. PVM is more oriented towards networks of workstations. It provides an abstraction of a processor pool. In fact, each processor in the pool can be separate network node(Rinat and Ahmed, 2012). The programmer can use PVM routines to start/stop a process or send message between processes. PVM exists for different architectures and a typical PVM program written for different architectures can communicate with each other thus allowing for building of heterogeneous networking computing systems.

2.4.3 A-BSP

A-BSP system is an implementation of Bulk Synchronous Parallel computation model. In A-BSP, workers are linked in a logical ring. Initial state of each process is replicated on several workers in the ring. When a worker finishes computation of one process, it sends results to its successor in the ring. If a worker does not get a message from it predecessor, it starts execution of the process of its predecessor using the replicated initial state.

2.4.4 OpenMP

OpenMP is an application interface (API) for parallel programming on multiprocessors. It consists of a set of compiler directives and a library of support functions. The APIs in OpenMP can be used in sequential programming languages like Fortran, C and C++.OpenMP stands at high level of abstraction which eases the

development of parallel applications from the perspective of the developer. OpenMP hides and implements by itself the details like work load partitioning, worker management, communication and synchronization(Ravela, 2010).

2.4.5 TBB

Threading building blocks is a parallel programming library developed by the Intel Corporation and designed to be used with C and C++ programming languages (Thread Building Block, 2013). It offers a very highly sophisticated set of parallel primitives to parallelize the application and to enhance the performance of the application on many cores. Intel threading building blocks is a high level library and supports the task-based parallelism to parallelize applications; it not only replaces the threading libraries, but also hides the details about the threading mechanisms for performance and scalability. Task creation and management are in general faster than threads. A typical task is 18 times faster than threads on a Linux based system and almost 100 times faster than a windows-based system. Intel TBB relies on offering the scalable data parallel programming which is much harder to achieve by making use of the performance as the number of processor cores increases. Threading building blocks is a library that supports the scalable parallel programming by using the C++ code and does not require any special languages or compilers. Threading building blocks provides the standard templates for common parallel iteration sequences to attain increased speed from multiprocessor cores without having to be experts in synchronization, load balancing, cache optimization. Programs using TBB will run on systems with a single processor, as well as on systems with multiple processor cores. Additionally it fully supports nested parallelism, so that you can develop larger parallel components form smaller components easily. To efficiently use the library, the developer has to specify tasks instead of probing into the library by managing

threads. The underlying library is responsible for mapping the tasks on to the threads in an efficient manner. As a result TBB enables you to specify parallelism more efficiently than using other models for scalable data parallel programming. In general as one goes deep into the TBB parallel programming model it is hard to understand the model and in some cases increases the development time.This is because TBB stands at a high level of abstraction(Ravela, 2010).

2.4.6 Cilk

Cilk is a multithreaded language for parallel programming that generalizes thesemantics of C by introducing linguistic constructs for parallel control.(Brown, 2010).This extension to C provides the programmer with simple constructs which theycan use to create a parallel program. The Cilk extension works from two angles,firstly during compilation the C is analyzed statically and calls to the Cilk parallellibrary are added into the C postsource which is generated (ready for compilationby a normal C compiler.) Secondly, during runtime, C code will call the Cilk parallellibrary which will actually support the parallelism(Brown, 2010). Cilk'sparallelism is limited to Symmetric Multiprocessors (SMP).

2.4.7 Cilk ++

The Cilk++ language extends C++ to simplify writing parallel applications that efficiently exploit multiple processors(Intel, 2009).The Cilk++ language is particularly well suited for, but not limited to, divide and conquer algorithms. This strategy solves problems by breaking them into sub-problems (tasks) that can be solved independently, then combining the results. Recursive functions are often used for divide and conquer algorithms, and are well supported by the Cilk++ language.

The tasks may be implemented in separate functions or in iterations of a loop. The Cilk++ keywords identify function calls and loops that can run in parallel. The Intel Cilk++ runtime system schedules these tasks to run efficiently on the available processors.

2.5 Wired and Wireless Computer Networks

2.5.1 Wired Networks

Wired Local Area Networks (LANs) have been in existence for quite some decades now with technologies like Ethernet, Token Ring, Tokin Bus, Fibre Distributed Data Interface (FDDI) and Asynchronous Transfer Mode (ATM) LAN.In order to setup standards for intercommunication among various manufacturer devices; the project 802 was introduced in 1985 by Institute of Electrical and Electronics Engineers (IEEE). The standard was adopted by the American National Standards Institute (ANSI) and in 1987, the International Standard Organization (ISO) also approved it as an international standard under the designation ISO 8802(Behrouz & Sophia , 2007). Since Ethernet is the network implemented in this research, there will be more emphasis on the Ethernet network in this literature.

Project 802 divides the Data Link layer of the OSI model into two sub-layers namely the Logical Link Control (LLC) and the Media Access Control (MAC).

2.5.2 Logical Link Control Sub –Layer

The LLC provides one single data protocol for all IEEE LANs, the protocol data unit (PDU) in the LLC takes care of data framing, flow control and error control.

2.5.3 Media Access Control Sub –Layer

The MAC provides different protocols for different IEEE LANs, that is, it provides different protocols for Token Ring, Ethernet LAN, FDDI and Token Bus.

2.5.4 Wired LAN: Ethernet

Ethernet has evolved through the years with the first Ethernet developed in1976 at Xerox's Palo Alto Research Center (PARC). Since then, Ethernet has gone through four generations namely, Standard Ethernet (10 Mbps), Fast Ethernet (100 Mbps), Gigabit Ethernet (1 Gbps) and Ten-Gigabit Ethernet (10 Gbps)(Behrouz and Sophia , 2007).

A. Standard Ethernet

That standard Ethernet is governed by Mac Sublayer which is basically in charge of framing the data**.** In this type of Ethernet, the access method is through Carrier Sense Multiple Access/Collision Detection (CSMA/CD) protocol. The choice of the minimum and maximum frame-length is to allow for smooth operation of the CSMA/CD protocol.

The physical layer of the standard Ethernet makes use of digital signal(baseband) at 100M Mbps). Data is encoded and decoded at the sender and receiver's end using the Manchester Schemes.

The common implementation methods for the Standard Ethernet are 10Base-S, 10Base-2 10Base-T, 10Base-F. These implementations methods specify the physical layer cabling.

The 10Base-S also known as the Thicknet or Thick Ethernet makes using of coaxial cables on a bus topology with possibility of collision hence the CSMA/CD is fully implemented. The coaxial cable in a 10Base-S has a maximum length of 500m and

the transmitter/receiver for a typical 10Base-S network is decoupled from the Station/Computer.

10Base-2 is also implemented on a bus topology but with a thinner and cheaper cable hence the name Thinnet or Thin Ethernet. Its transmitter/receiver is built with the Network Interface Card (NIC). However, the maximum cabling length for the Thin Ethernet is 185m and it implements the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol because of the possibility of collisions.

100Base-T introduces a networking device known as the hub. It makes use of a twisted pair of cables, one for transmitting and the other for receiving. The physical setup of this type of Ethernet made sure that the only collision that could occur would be at the hub, hence the implementation of CSMA/CD. The maximum length of a twisted pair is 100m.

10Base-F:Although there are several types of optical fibre 10 Mbps Ethernet, the most common is called 10Base-F. 10Base-F uses a start topology to connect stations to a hug. The stations are connected to the hug using two fibre-optic cables.

B. Fast Ethernet

Fast Ethernet was designed to compete with LAN protocols such as FDDI or Fiber Channel (or Fibre Channel, as it is sometimes spelled). IEEE created Fast Ethernet under the name 802.3u(Behrouz and Sophia , 2007). The main consideration for the fast Ethernet was to leave the Mac sub-layer untouched but to drop the bus topology of the Standard Ethernet. However, there is a choice between Full-Duplex and Half Duplex mode for as the access method. There is no implementation of CSMA/CD for full-duplex though fast Ethernet supports backward compatibility. This research makes use of the Fast Ethernet setup.

Fast Ethernet implementation at the physical layer can be categorized as either two-wire or four-wire. The two-wire implementation can be either category 5 UTP(100Base-Tx) or fibre-optic cable (100Base-FX). The four-wire implementation is designed only for category 3 UTP (l00Base-T4).

In terms of topology, the gigabit Ethernet is able to connect two or more stations. When the number of stations connected is more than two then a switch device is used to implement a start topology. If there are two stations then they can be connected point-to-point

C. Gigabit Ethernet

Gigabit Ethernet's design as contained in the IEEE committee's 802.3a standard is among other things designed to upgrade to a data rate of 1Gbps and support the features already present in the fast Ethernet. The Mac sublayer of the gigabit Ethernet layer was left untouched however, gigabit Ethernet is mostly implemented on a full-duplex mode network(Behrouz and Sophia , 2007) and since there is no collision in this mode the CSMA/CD protocol is not used. In terms of topology, the gigabit Ethernet is able to connect two or more stations. When the number of stations connected is more than two then a switch device is used to implement a start topology. If there are two stations then they can be connected point-to-point

2.5.4 Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

This is a media access method in which a station senses the medium for idleness and transmits data packets.If at the same time another station transmits then a collision happens and both stations backs off for a random amount of time before retransmitting.

Behrouz and Sophia (2007)states that the basic idea behind CSMA/CD is that, a station needs to be able to receive signalswhile transmitting a signal to detect a collision. When there is no collision, the station receives one signal: its own signal. When there is a collision, the station receives two signals: its own signal and the signal transmitted by a second station. To distinguish between these two cases, the received signals in these two cases must be significantly different. In other words, the signal from the second station needs to add a significant amount of energy to the one created by the first station.

In wired network, the received signal has almost the same energy as the sent signal because either the length of the cable is short or there are repeaters that amplify the energy between the send and the receiver. This means that in a collision, the detected energy almost doubles.

CSMA/CD is a modification of pure carrier sense multiple access (CSMA). CSMA/CD is used to improve CSMA performance by terminating transmission as soon as a collision is detected, thus shortening the time required before a retry can be attempted.

CSMA/CD was used in now obsolete shared media Ethernet variants (10BASE5, 10BASE2) and in the early versions of twisted-pair Ethernet which used repeater hubs. Modern Ethernet networks built withswitches and full-duplex connections no longer utilize CSMA/CD though it is still supported for backwards compatibility. IEEE 802.3, which defines all Ethernet variants, for historical reasons still, bears the title "Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications"(wikipedia, Carrier Sense Multiple Access With Collision Detection, 2013).

2.5.5 Wireless Networks

A wireless network is any type of network that is not connected by cables of any kind. Wireless telecommunications networks are generally implemented and administered using radio communication. This implementation takes place at the physical layer of the OSI model network structure(wikipedia, Wireless Networks, 2013). Types of wireless networks include, Wireless PAN, Wireless LAN, Wireless MAN, Wireless WAN and Cellular networks.

Wireless Personal Area Network (Wireless PANs) interconnects devices within a relatively small area, that is generally within a person's reach. Examples of these types of networks are Bluetooth radio networks, infra-red networks.

Wireless Local Area Networks (Wireless LANs) connects two or more devices over a short distance using a wireless distribution method, usually providing a connection through an access point. WLANs form an integral part of this research as such a detailed description will be given shortly.

Wireless Metropolitan Area Network (Wireless MAN) are a type of wireless networks that connects several wireless LANs. A typical example is the WiMAX and is described by the IEEE 802.16 standard.

Wireless Wide Area Networks (Wireless WANs) are wireless networks that typically cover large areas, such as between neighbouring towns and cities, or city and suburb. These networks can be used to connect branch offices of business or as a public internet access systems. The wireless connections between access points are usually point to point microwave links using parabolic dishes on the 2.4GHz band, rather than omnidirectional antennas used with small networks.

Cellular Networks or mobile networks are radio networks distributed over land areas called cells, each served by at least one fixed-location transceiver, known as a cell site or base station. Each cell in this network uses a different set of radio frequencies from all their immediate neighbouring cells to avoid any interference.

2.5.6 Wireless Local Area Networks (WLANs)

The Institute of Electrical and Electronics Engineering (IEEE), European Telecommunication Standard Institute (ETSI) and Home Radio Frequency Working Group (HomeRFWG), have been involved in developing standards for the WLAN. These include the IEEE 802.11x, HiperLANx, and HomeRG. Of these, the IEEE 802.11 family of protocols has clearly become the dominant standard for WLAN in the world(Dharma and Zeng, 2011).

A. IEEE 802.11

The IEEE 802.11 is a set of standards for wireless area networks and these standards cover the physical and data link layers of the OSI model.The major protocols include IEEE 802.11a, IEEE 802.11b, IEEE 802.11g and IEEE 802.11n. Their most significant differences lie in the specification of the physical layer(Dharma and Zeng, 2011).

B. IEEE 802.11a

Ratified in 1999, the IEEE 802.11a operates on 5GHz frequency band and has a throughput of 54Mbps. This specification has less interference, since the 2.4GHz band is heavily used. However, its penetration is also reduced, due to its higher carrier frequency, so the signals are absorbed readily by solid objects along its propagation path(Dharma and Zeng, 2011).

IEEE 802.11a uses Orthogonal Frequency-Division multiplexing (OFDM) with 52 subcarriers spanning over a 20MHz spectrum.

C. IEEE 802.11b

In August 1999, a group of industry leaders formed a nonprofit organization calledthe wireless Ethernet compatibility alliance (WECA) to promote the IEEE 802.11high-rate standard (which eventually became IEEE 802.11b) as a commercial standardto ensure the interoperability of different vendors' products. WECA selectedan independent test lab to test and certify the interoperability of the IEEE 802.11bproducts. IEEE 802.11b operates on 2.4GHz band with throughput of up to 11Mbps, which was released in 1999 and was marketed under the nameWi-Fi. IEEE802.11b uses a direct extension of direct-sequence spread spectrum DSSS on thePHY layer. DSSS uses a continuous string of pseudonoise (PN) code symbols tomodule information, which allows multiple transmitters to share the same channelwith orthogonal PN codes. WECA was later renamed the Wi-Fi alliance and certifiesall the IEEE 802.11 high-rate standards (which include the IEEE 802.11b,IEEE 802.11a, and IEEE 802.11g) products. Almost all companies selling theIEEE 802.11 equipment are members of the Wi-Fi alliance.

D. IEEE 802.11g

Released in 2008, it is the third modulation standard for WLAN. It operates on 2.4GHz like IEEE 802.11b. The physical layer can use either Direct-Sequence Spread Spectrum (DSSS) or Orthogonal Frequency Division Multiplexing (OFDM). It can achieve higher throughput of up to 54Mbps due to its heritage from the IEEE 802.11g(Dharma and Zeng, 2011).

E. IEEE 802.11n

IEEE 802.11n is the recent amendment that incorporates multiple-input multipleoutput(MIMO) technology. The bandwidth in IEEE 802.11n can be 40MHz, andthe maximum physical layer data rate is raised from 54Mbps to an objective of up to600Mbps. MIMO improves communication performance with the use of multipleantennas at both the transmitter and receiver for multiple transmitted data streams.Significant increase in data throughput and link range can be observed in applyingMIMO, without additional cost of bandwidth or transmission power, benefitingfrom antenna diversity and spatial multiplexing. IEEE 802.11n can operate on2.4GHz and 5GHz. It uses either DSSS or OFDM for physical layer modulation(Dharma and Zeng, 2011).

2.5.7 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)

This is a scheme used in detecting and avoiding collision in wireless networks. In this CSMA/CA stations try to avoid collisions by only transmitting when medium is idle.

Packets delivery in IEEE 802.11 is asynchronous and connectionless therefore, it employs a best effort approach by default unless quality of service turned on(Dharma and Zeng, 2011).

The Media Access layer of the IEEE 802.11 employs methods for transmission namely Distributed Coordination Function (DCF), Point Coordination Function (PCF), or Hybrid Coordination Function (HCF)(Dharma and Zeng, 2011).

DCF is carrier sense multiple access with collision avoidance and a random backoff intervals is used for the determination of the defer-period after collision is sensed.

PCF is an option access method, which is only for configuration of the infrastructure network. In the operation of PCF, the infrastructure polls every station to determine

the sequence oftransmission. The information in PCF is distributed by beacon management frames.As the infrastructure controls the transmission, contention can be mitigated to someextent.

HCF, combines functions from DCF and PCF with enhancement on quality of service.

CSMA/CA can optionally be supplemented by the exchange of a Request to Send (RTS) packet sent by the sender S, and a Clear to Send (CTS) packet sent by the intended receiver R. Thus alerting all nodes within range of the sender, receiver or both, to not transmit for the duration of the main transmission. This is known as the IEEE 802.11 RTS/CTS exchange. Implementation of RTS/CTS helps to partially solve the hidden node problem that is often found in wireless networking(Wikipedia, 2013)

2.6 Review of Related Works

Over the years, comparisons have been made on various subjects bordering parallel programming but as at the time of this research no empirical analysis have been conducted on the performance of parallel programs on both wired and wireless LAN.

Ravela(2010) compared the performance of shared memory parallel programs like TBB, Pthreads, Cilk++ and OpenMP.Though an empirical analysis was carried out, it was based on comparing shared memory parallel computing libraries.

Rinat and Ahmed (2000)surveyed and analysed existing frameworks for network computing with a particular focus on computing systems for Network of Workstations(NOW).

Arnab and Nabanita, (2009) compared performance of broadcast with point-to-point communication (both blocking and non-blocking) of the MPI-I standard library on a

cluster computer in communicating the same data block among all processing node. They compared the performance in terms of delay by varying the number of processing nodes and the data block size.

Fernando(2004) analyzed problems and possible solutions taking into account the main factors of computing and communications in parallel programs installed on local area networks.

In all the related works found and sited, none of them discussed the empirical performance of parallel programs on wireless LANs as compared to programs on wired LANs. The viability and scalability of parallel programs on wireless clusters has never been measured empirically. Therefore, the need for this research.

2.7 Summary

This chapter has highlighted the various literatures in parallel computing**.** The various types of parallel computers, parallel programming models, parallel libraries/programming languages and a review of works related to this research have been discussed in detail.

CHAPTER THREE

PARALLEL MATRIX-VECTOR MULTIPLICATION EXPERIMENT

3.1 Introduction

This chapter contains details of how the parallel matrix-vector multiplication experimentsfor this empirical analysis werecarried out. For the purpose of comparison, it gives detailsof thedesign and algorithm time complexity analysis of each of the parallel matrix-vector algorithms used as benchmark program in this research.It also discusses results obtained from each of the parallel matrix-vector multiplication algorithms analysed.

3.2 Experimental Setup

3.2.1 System Configuration

In this experiment, a network is setup containing ten (10) laptops all with the same hardware configuration. The choice of using laptops for this research is purely for convenience because the same experiment would have been carried out with desktop computers.Each of the laptops has a32 bit Pentium® Dual-Core T4200 2.00GHZ 2.00 GHZ , 2.0 GB Random Access Memory, Realtek RTL8187B Wireless 802.11b/g 54mbps wireless adaptor and PIC-E Gigabit Ethernet Network Interface Card.

3.2.2 Network Setup

The Topology for both wired and wireless local area networksis star. For the wired network, a fast Ethernet is setup composing of a 24 Port DLink switch and a category five Ethernet cable use to connect the laptops.

The wireless network was setup using a DLink wireless Access point. The wireless network setup conforms to the IEEE 802.11g standard hence has the maximum throughput of 54Mbps. A secured Service Set Identification (SSID) was setup for connecting the laptops wirelessly. On each laptop is a wireless 802.11b/g adapter with a maximum speed of 54mbps.

3.2.3 Operating System

All the laptops were installed with Microsoft Windows 7 Operating System. It should be noted that the Windows firewall and antivirus for each laptop was switched off in order to avoid denial of access among the laptops. The firewall and antivirus could have been configured to include exceptions for the intended programs but for avoidance of complexity they were switched off.

Also, each laptop was setup never to hibernate or sleep when idle. This was to make sure all laptops remain active during program execution.

3.2.4 Parallel Library Setup (Using DeinoMPI)

All the laptops were installed with the 32bit version of DeinoMPI. On each laptop, a user with username ‘mpiuser’ and password ‘mpiuser’ was setup at the operating system level. This is because DeinoMPI’s master process logs into every computer in the cluster to start a process during parallel computation and to form a grid of one massive parallel computational environment the username and password must be uniform.

It should also be noted here that the username or password used in this experiment are changeable hence any combination of username and password can be used.What is

most important is that all systems have the same username and password for communications to be effective.

At the DeinoMPI level, each laptop is required to have a credential store that can be tied to the system's registry. And a DeinoMPI user and password was created. This user and password is similar to the username and password created at the operating system's level.

For programs to be executed in parallel, DeinoMPI process manager needs to know the location of the program on each laptop to executive the program. Windows 7 can support only a limited number of simultaneous connections to a file, and since in this experiment, multiple processes will have simultaneous connection to a file; each copy of the program was copied into each of the laptops. Therefore, a folder called "MPI_Simulation" was created on each laptop. In this folder, the programs to be executed along with the parameter files were copied.

3.2.5 Program Compilation

Several C compilers were used to compile the programs. A combination of gcc compiler-which comes with Cygwin- and Dev++ were used to compile the programs. As will be discussed shortly, it is observed that for jumpshots data to be generated, the program had to be compiled with a compiler pre-designed for Windows Operating System. However, the **gcc** compiled programs worked better; the only disadvantage was that jumpshot data could not be generated with a Cygwin-gcc compiled program.The reason was that the programstend to behave like it is in a linux environment by trying to locate the external libraries for the jumpshot whose directory location is different in a windows environment. The C programs can be found in appendix A of this work.

3.2.6 Execution Plan

All programs were executed using Windows shell script. An execution plan was setup and for each case study(wired or wireless),Windows shell script was written to run the already compiled program. The Windows shell script can be found in the appendix of this work.

It should be noted that a separate execution plan with a separate shell script was written for multiprogramming scenario of this experiment.

All Windows shell script codes can be found in appendix of this thesis.

3.3 Sequential Matrix – Vector Multiplication Algorithm

Given a matrix A with dimension $m\ x\ n$ and a column vector b with dimension $n\ x\ 1$, a sequential matrix-vector multiplication algorithm multiplies each row of matrix A with the column vector b. Therefore, the algorithm requires two loop nested loops. The outer loop traverses the rows of A while the inner loop traverses the columns of A whose size equals the rows of the vector b. In general for any matrix A of dimension $m\ x\ n$, performs n multiplication m times, requiring a running time complexity of $\theta(mn)$. For a square matrix with dimension $n\ x\ n$, the running time complexity becomes $\theta(n^2)$.

3.4 Parallel Matrix-Vector Multiplication Algorithm

The main aim of the parallel matrix-vector algorithm is to decompose (divide) computations among processing nodes and allow each processing node to simultaneously compute its own portion of the matrix. There are various decomposition techniques namely, row-wise decomposition, column-wise decomposition and checkerboard decomposition.

Each of these decompositions come with its each communication and computation bottlenecks, hence their speed is likely to vary according to the decomposition method used.

In this analysis, all three decomposition techniques were executed on both wired and wireless networks to observe their performance differences as related to the network used. The sole reason for picking these algorithms is to have different bases for comparison of performance. The well-known sequential Matrix-Vector Multiplication algorithm runs at $\boldsymbol{\theta}(n^2)$time complexity for square matrix $n\,x\,n$ and $\boldsymbol{\theta}(mn)$time complexity for matrix of dimension $m\,x\,n$, both with vector b of dimension $n\,x\,1$. The time complexity for the parallel matrix-vector multiplication algorithm is dependent on factors such as the communication complexity and the decomposition complexity. The time complexity of each of the parallel matrix-vector algorithms will be discussed shortly.

3.4.1 Parallel Matrix-Vector Multiplication Algorithm (Row-wise Decomposition)

The goal of every parallel program is to find a way to speed up computation. Given an $n\,x\,n$ Matrix A and an $n\,x\,1$ Vector b, row-based decomposition algorithm divides Matrix Asuch that, if there are p number of processes then each process is allocated at most⌈n/p⌉rows of the $n\,x\,n$ matrix A. The $n\,x\,1$ vector b is distributed across all processes in the communication grid.

In the row-based matrix-vector multiplication, domain decomposition is done such that each row of the matrix is associated with the entire vector b. This is because it has only n elements hence decomposing it will lead to additional and unnecessary communication. The Row-wise decomposition is as shown in Figure 3.1

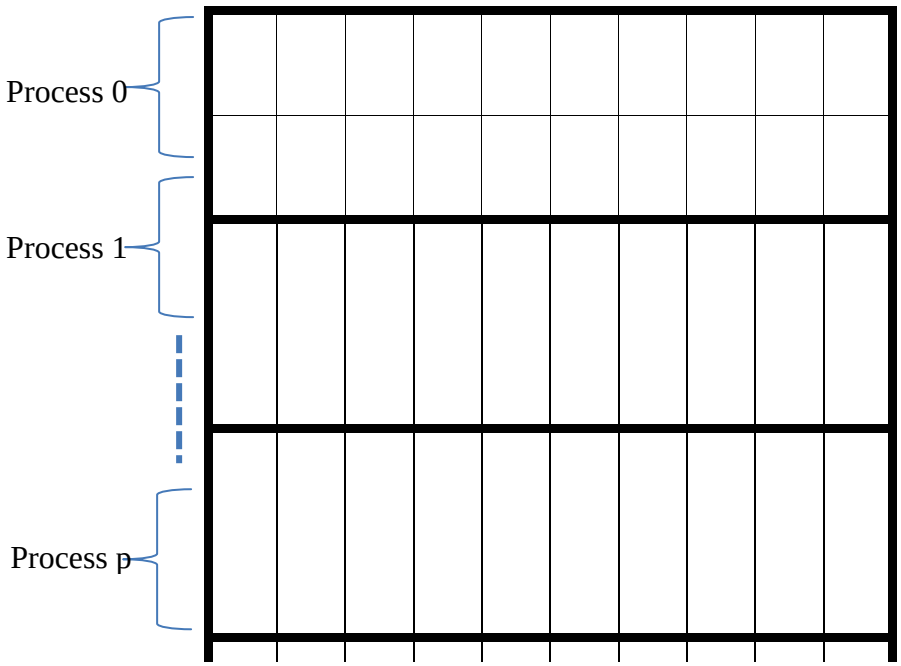


Figure 3.1: Row-wise decomposition of matrix

The algorithm design steps follows the Ian Forster's parallel algorithm design steps (Michael, 2003).

Row-wise Partitioning and Primitive Task

The primitive task involves multiplying each row i of Matrix A by the vector c producing an element ci of the resultant vector c as illustrated in Figure 3.2

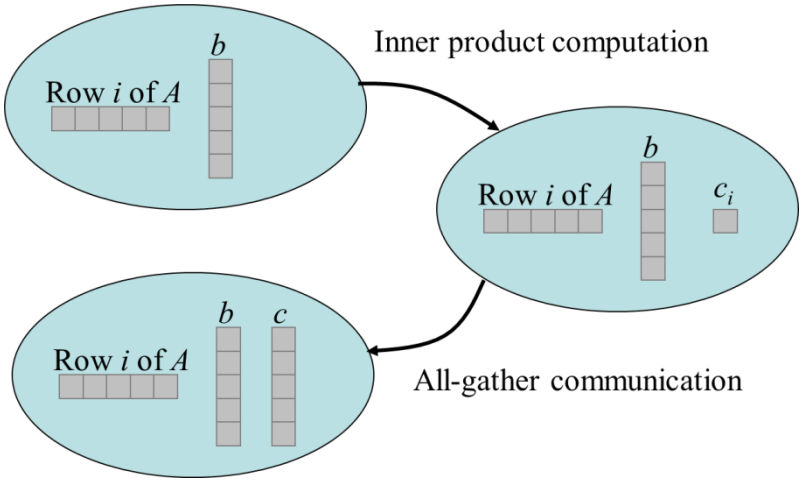


Figure 3.2:Primitive task involving the multiplication of row i of A and vector b

Communication Step

The Communication step involves an MPI_Allgather operation as illustrated in the figure 3.2. This is so as to collect all the computed elements of the resultant vector c into the final resultant vector c.

MPI_Allgather concatenates blocks of vector distributed among a group of processes and copies the resulting whole vector to all the processes in the communication group. A communication group is a collection of processes together communicating to solve a task.

MPI_Allgatherv is similar to MPI_Allgather.However, it is used when the number of elements contributed by each process to the resulting concatenated vector varies.

MPI_Allgather operation is required in this algorithm to concatenate all the partial product of each process into one single resultant vector c.

In this algorithm, the number of elements per process is only a constant if the number of elements is a multiple of the number of processes. This is always not the case in this algorithm so an MPI_Allgatherv is used.

Agglomeration and Mapping

Obviouslynumber of participating processes is likely to be less than or greater than the number of rows in the Matrix A. Therefore, agglomeration and mapping of the primitive task is required. Each process is allocated with at most $\lceil n/p \rceil$ rows of Matrix A, hence each process performs its own portion of the computation independently.For each $c_{i,1}$ element of result vector c, $c_{i,1}=\sum_{j=0}^{y-1} a_{i,j} b_{j,1}$, where a and b are elements of $n\ x\ n$ Matrix A and $n\ x\ 1$ vector b respectively, y is the number of rows received by process i.

After computation, each process produces a group of elements (based on the number of rows of Matrix Ait received) of the resultant vector$c$. The algorithm then performs an MPI_Allgatherv operation which collects the group of results elements of vector c into a resultant vector c, hence the need to allocate memory for the resultant vector c on each process.

Analysis

The sequential matrix-vector multiplication algorithm takes $\boldsymbol{\theta}(n^2)$. In the parallel version, the first step in this algorithm is the computation of the received portion of A by each process. Each portion is allocated not more than $\lceil n/p \rceil$ therefore this step takes$\boldsymbol{\theta}(\frac{n^2}{p})$.

After computation, an MPI_Allgatherv operation is performed. For an efficient MPI_Allgatherv operation, $log\ p$ messages need to be sent, where p is the number of processes and the total number of elements passed is$\frac{n(p-1)}{2}$, when p is a power of 2.Putting everything together, the parallel algorithm runs at $\boldsymbol{\theta}\left(\frac{n^2}{p} + \log p + \frac{n(p-1)}{2}\right)$

3.4.2 Execution Plan

The parallel matrix-vector (row-wise decompostion) algorithmwas executed on windows platform with all the participating systems running DeinoMPI. Therefore, there was need for windows batch script to help simulate multiple runs of execution for the programs.

Windows 7 only allows for 20 concurrent connections to a file (Microsoft Corporation,2008), and since this simulation at certain points in time will involve

more than 20 processes to access the matrix file, all the executable programs plus the matrix and vector files had to be copied to every system partaking in the MPI computation.DeinoMPI supports this mode of execution with the condition that the director structure to the program must be the same on all the systems partaking in the MPI computation.

This setup was done to cater for the future execution of programs that later required the execution of more than twenty process as will be seen in multi-programming approach of parallel matrix-matrix multiplication algorithm which will be discussed in chapter four of this thesis.

The Jumpshots clog files produced after each simulation were converted to slog files using the Jumpshot program that comes shipped with DeinoMPI installation.

The windows scripts for this thesis are provided in appendix.

3.4.3 Results of Parallel Matrix-Vector Multiplication Algorithm (Row-wise Decomposition) on Wired and Wireless LAN

In this experiment, a $4000\ x\ 4000$ matrix whose elements are of data type double was multiplied with a $4000\ x\ 1$ vector whose elements are also of data type double. A total of 100 runs were made. Ten (10) runs were made for each number of clusters. After each cluster runs, the average execution time was obtained. The experiment was carried out on two (2) to ten (10) clusters. Figure 3.3 shows the outcome of

thisexperiment.

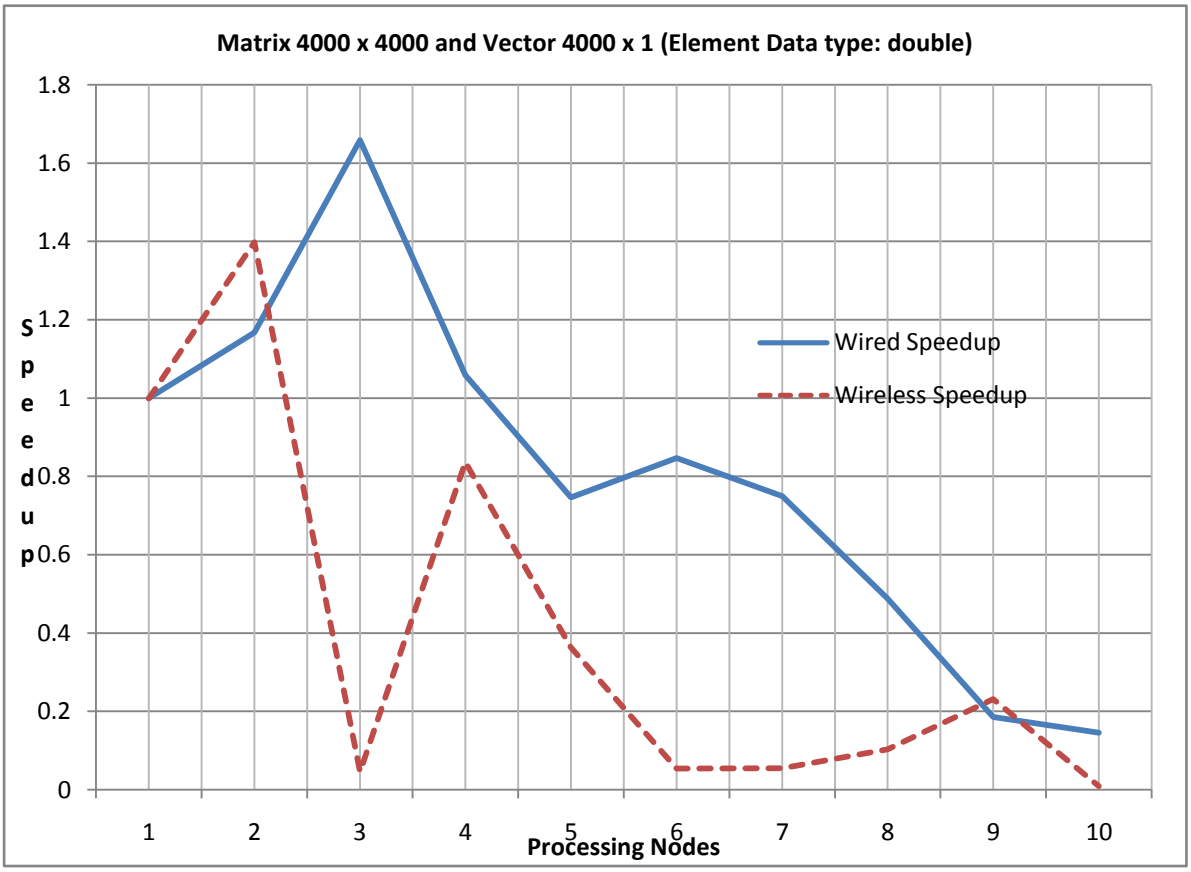


Figure 3.3: Performance for matrix-vector multiplication using row-wise decomposition

As can observed in the Figure 3.3, the degradation of performance in the wireless network starts occurring at the point when there are more than two processing nodes. Also, in this algorithm, there are more communication overheads than computational overhead and since each node/computer is relatively fast, the speedup for both steps tend to drop faster in this case it starts at processing node size 3). One of the reasons for the unwanted degradation in the wireless setup is in the fact that, in the wireless network, there is contention for the media resulting to collision. The introduction of back-off timer, contributes immensely to the behaviour.

This can further be seen the Jumpshot views in Figures3.4 to 3.7. Figures 3.4 and 3.5 show the performance of the matrix-vector multiplication (row-wise decomposition)

on two and three processing node size respectively. As the number of nodes increase, the number of rows allocated to each processing nodes is expected to decrease and if there is less delay and no collision in the network, it is expected that the amount of time required to send and receive data chunk should decrease since each chunk will then be smaller. This can be seen in send and receive time for Figures 3.4 and 3.7 respectively.

However, Figures 3.6 to 3.7 exhibit different behaviour. While the average send and receive time observed in the wired case decreased observed in Figures 3.4 and 3.5, it increased in Figures 3.6 and 3.7 respectively. It is obvious that the delay in send and receive increased in the wireless LAN as the node increases forcing a sharp decline in speedup as observed in Figure 3.3. Comparing Figures 3.4 with Figure 3.6, and Figure 3.5 with Figure 3.7, it is clearly seen that the speed of transmission vary clearly thereby affecting the speed of the overall algorithm.

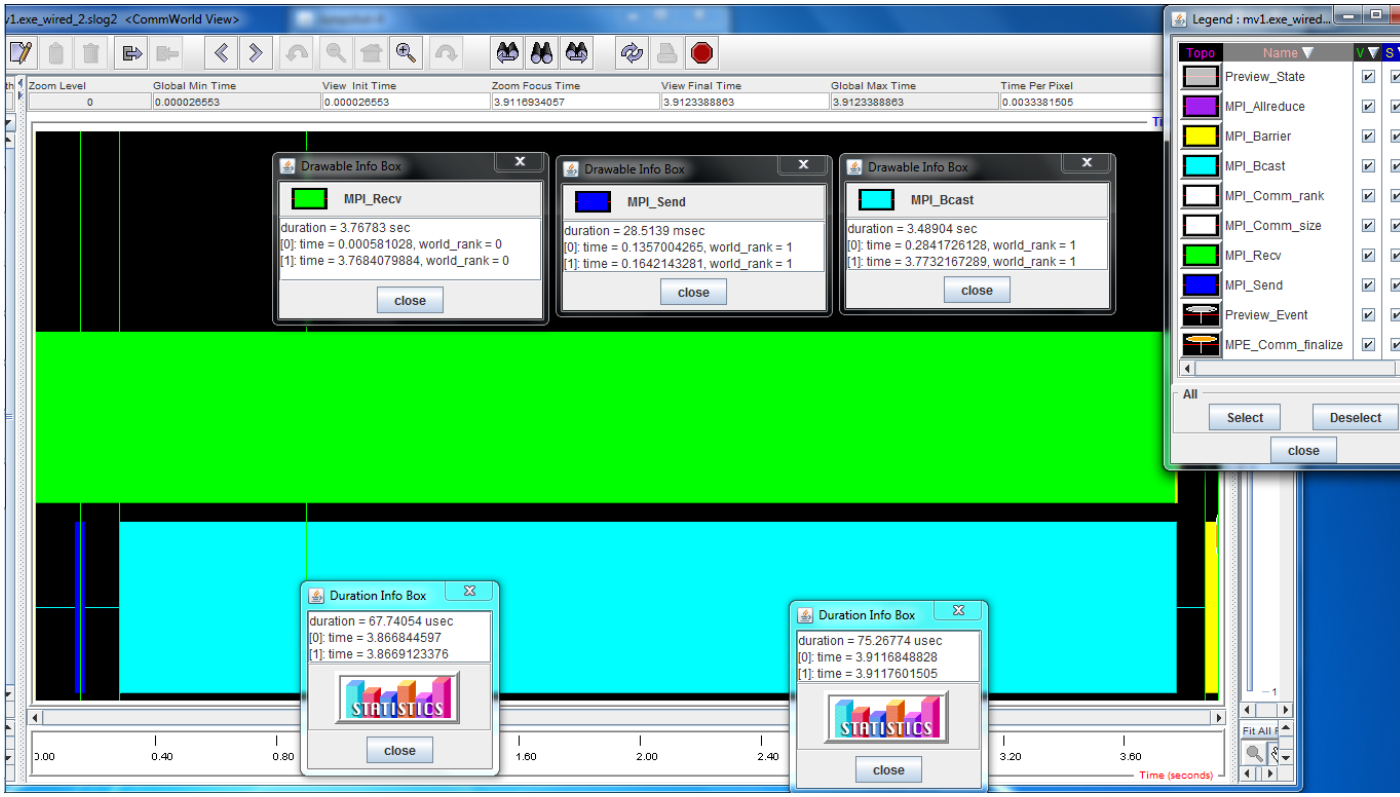


Figure 3.4: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition) on 2 Cluster wired LAN

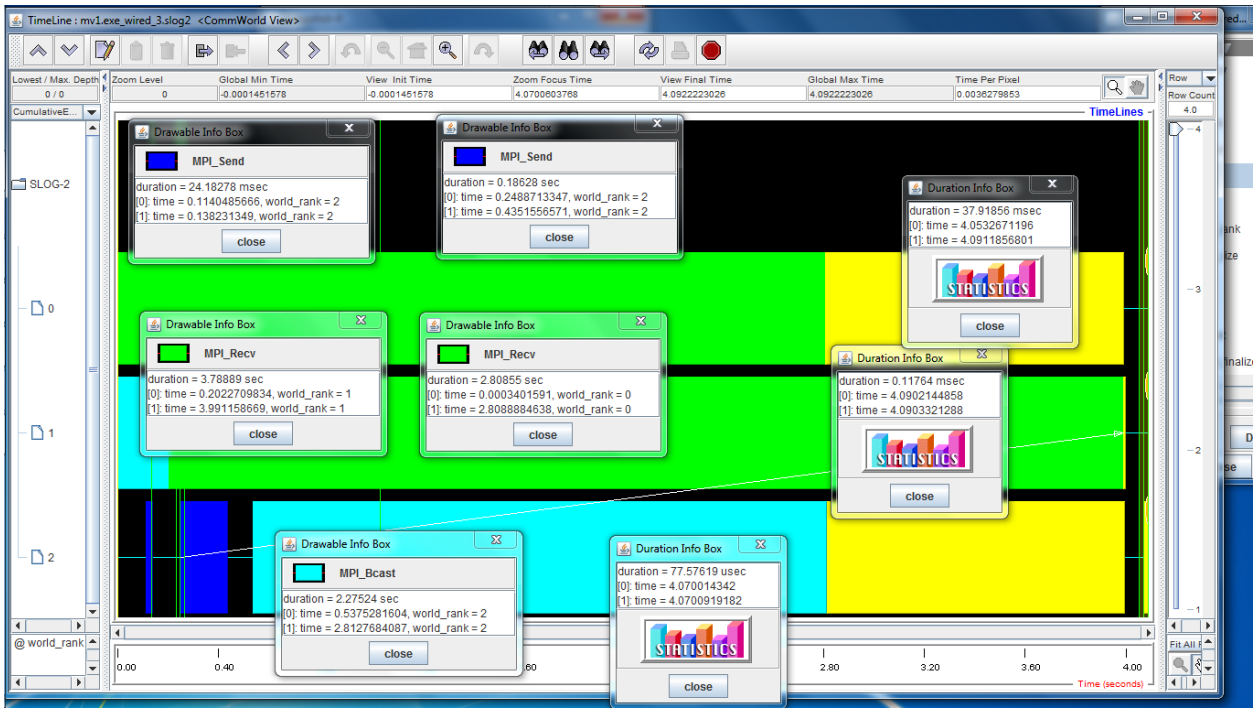


Figure 3.5: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition) on 3 Cluster wired LAN

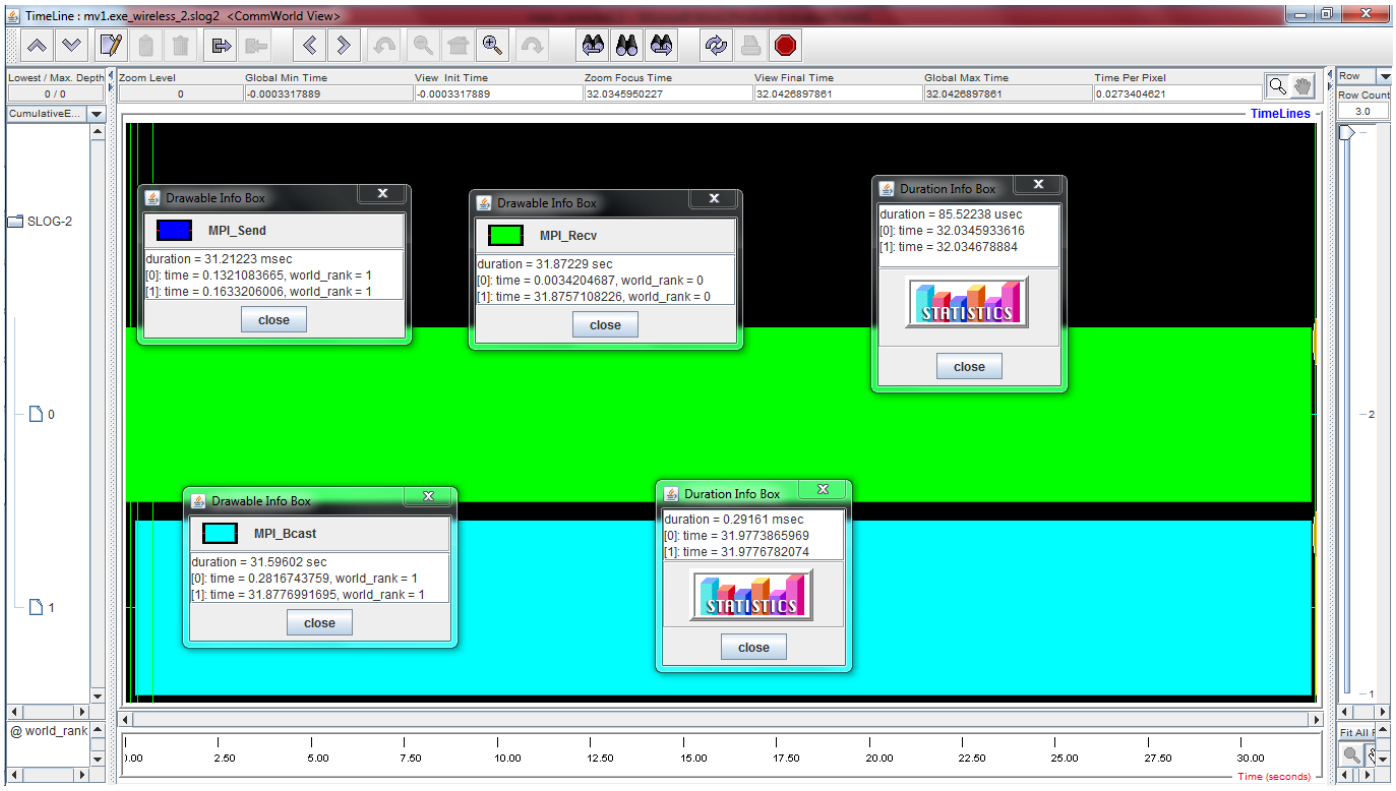


Figure 3.6: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition)

on 2 Cluster wireless LAN

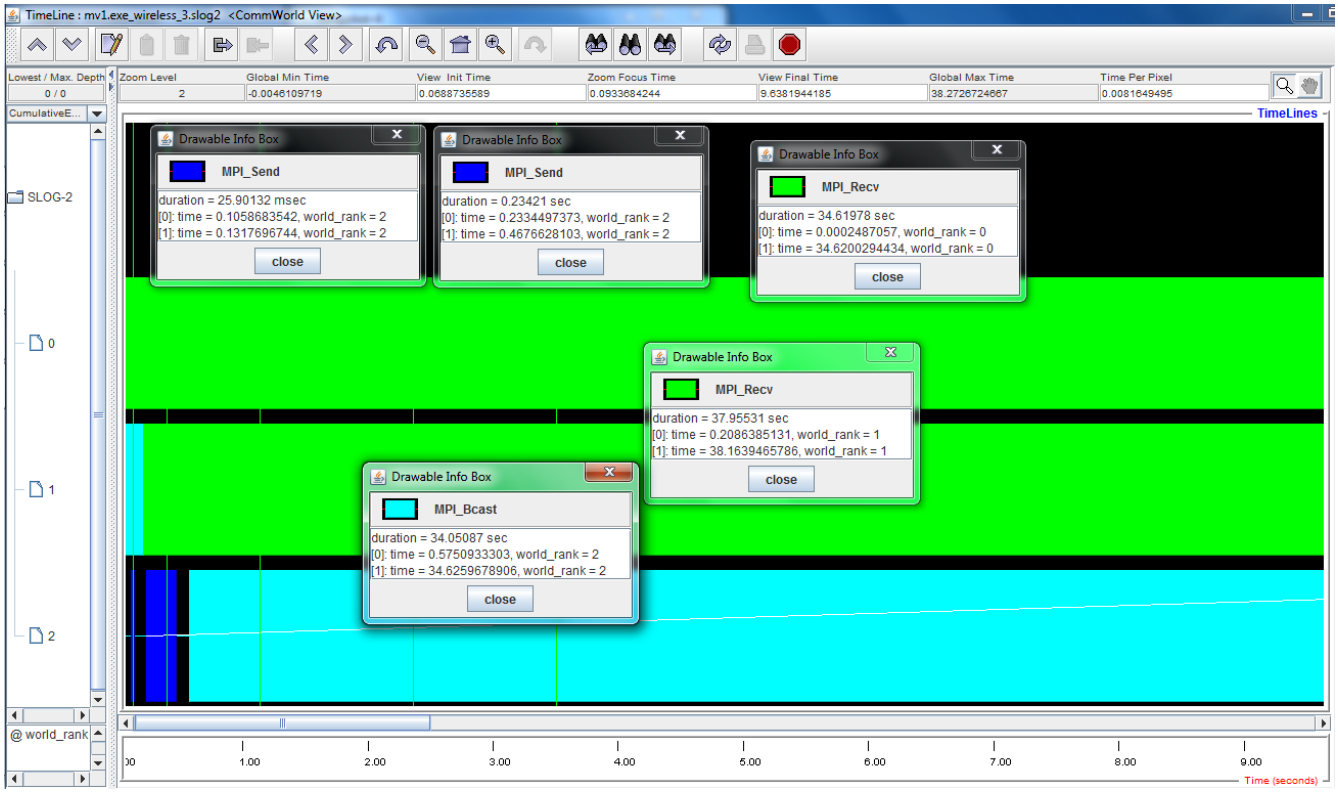


Figure 3.7: Jumpshot view of parallel matrix-vector multiplication (row-wise decomposition) on 3 Cluster wireless LAN

3.4.4 Parallel Matrix-Vector Multiplication Algorithm (Column-wise Decomposition)

The parallel Matrix-vector multiplication algorithm using Column-based decomposition has a similar concept with the row-based decomposition, except that matrix decomposition is column-wise.

In this algorithm a domain deposition of the matrix based on its columns is done, hence its columns are distributed among the various MPI processes participating in the communication. The domain decomposition can be foundin Figure 3.8

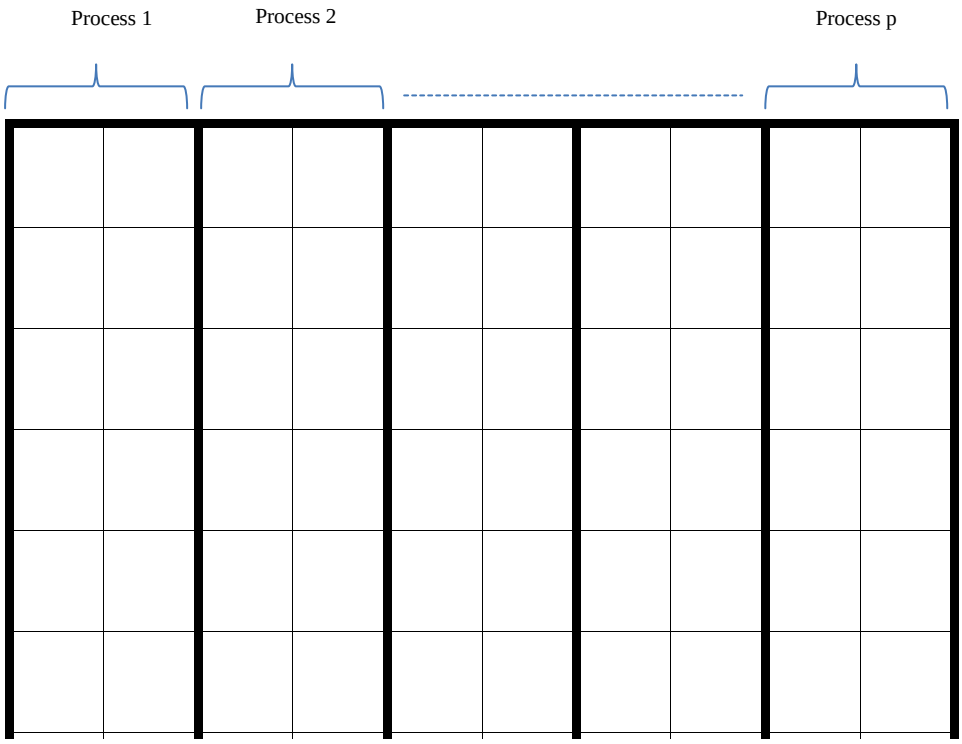


Figure 3.8: Column-wise decomposition of matrix

Column-wise Partitioning and Primitive Task

Given a matrix A of dimension $n\ x\ n$ and a vector b of dimension $n\ x\ 1$, each process is allocated not more than $\lceil n/p \rceil$columns of Matrix A. In a typical row-major matrix array data structurewhose elements need to be decomposed in a columwise manner, the elements of the matrix are NOT stored contiguously. In fact, each row of the matrix has to be redistributed across the processes partaking in the computation, hence an MPI_Scatter function is needed to redistribute a stripped row to the processes participating in the communication. This in effect simulates stripping the matrix column wise.

MPI_Scatteris a collective communication operation that performs a scatter: a group of elements held by the root process is divided into equal-sized chunks, and each chunk sent to every process in the communication group

After column-wise decomposition or partitioning, the primitive tasks involves multiplying a striped column$i$ of A with an element of b of vector b producing a

partial inner product vector $c\sim$. Thereafter, each task i needs only a single element of the result vector c_i. An MPI_Alltoall operating is then needed so that every partial result element j on task i must be transferred to task j. Finally a sum reduction (MPI_Reduce)operation is then performed on the n partial vector $c\sim$producing c_i.

MPI_Alltoall operation performs an all-to-all exchange within a communication group. Each process sends(and receives) the same number of elements to (and from) every process, including itself.

MPI_Reduce operation performs count reductions. When the function returns, the process with the rank root has the results of the reductions. Therefore a sum reduction is the sum of all the data elements whose result is relayed to the root process.

The matrix-vector multiplication (column-wise decomposition) algorithm is illustrated in Figure 3.9

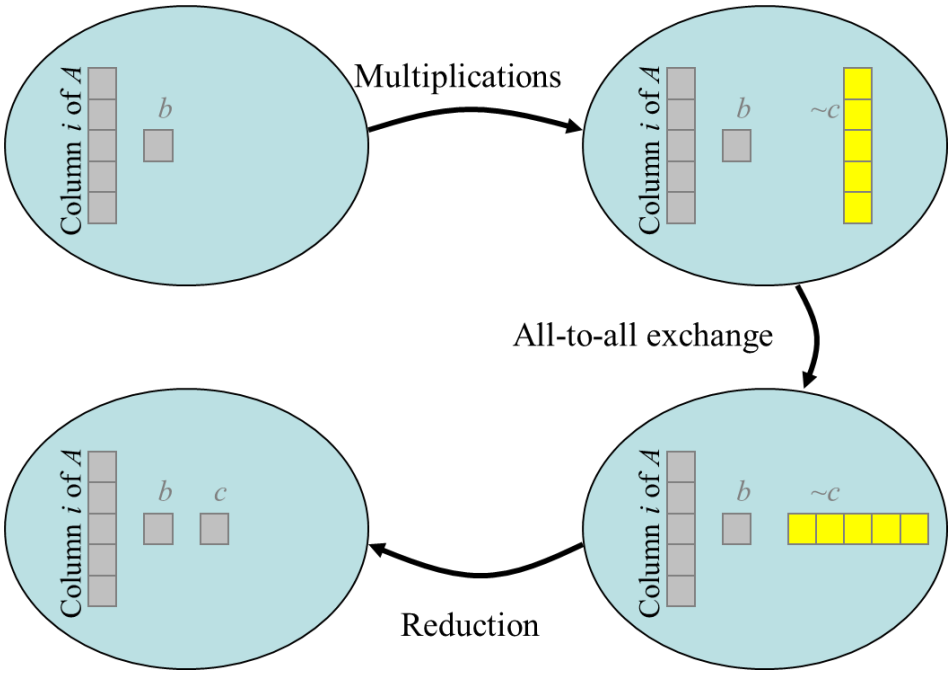


Figure3.9: Primitive task involved in a parrallel column-wise decomposed matrix-vector multiplication algorithm

Agglomeration and Mapping

Just like in the case of row-wise decomposition, the number of columns allocated to a process is at most $\lceil n/p \rceil$.Therefore, it is most likely that primitive tasks have to be combined and mapped to a single process.

Communication

After the primitive task, agglomeration/mapping and the computation phase, each process partaking in the computation performs an MPI_Alltoall function call. This is so as to retrieve all the corresponding inner product element c_i of the resultant vector c. An MPI reduction operation is then performed to sum all the inner product of $c_{i.}$ Finally an MPI_Gatherv vector function call is made to produce the resultant vector c.

MPI_Gatherv is a collective communication function that performs a gather operation. The root process gathers elements from every process i in a contiguous group of elements beginning with the element displacement. When the function returns, the result, a concatenation of elements is accessed through the receive buffer pointer of the function definition.

Analysis

Assuming m=n, each process multiplies its portion of matrix A by its block of vector b. No process is responsible for more than $\lceil n/p \rceil$ columns of A or elements of b. Hence the initial multiplication phase has time complexity $\boldsymbol{\theta}(\mathrm{n}\left(\frac{\mathrm{n}}{\mathrm{p}}\right)) = \boldsymbol{\theta}(\frac{n^2}{p})$. After the all-gather step, each processor sums the partial vectors collected from the other processes. There are p partial vectors, each of length at most $\lceil n/p \rceil$. The time complexity of this step is $\boldsymbol{\theta}(n)$. therefore, the overall computational complexity of the parallel algorithm is $\boldsymbol{\theta}(\frac{n^2}{p})$.

An all-to-all exchange can be performed in $\lceil n/p \rceil$ steps, using a hypercube communication pattern. During each step every process sends $n/2$ values to its partner and receives n/2 values from its partner. The total number of elements sent and received in the all-to-all exchange is $\lceil n/p \rceil$. Hence the communication complexity of this implementation of all-gather is $\boldsymbol{\theta}(n \log p)$.

Another way to perform an all-to-all exchange is for each process to send a message to each of the other $p-1$ processes. Every message contains just those elements the destination process is supposed to receive from the source. In this implementation the total number of messages is $p-1$, but the total number of elements passed by each process is less than or equal to n. the communication complexity of this algorithm is $\boldsymbol{\theta}(p+n)$.

When combined the computational portion of the algorithm with the final all-gather communication step, the overall complexity of the parallel matrix algorithm is either $\boldsymbol{\theta}(\frac{n^2}{\mathrm{p}} + \mathrm{n} \log p)$ or $\boldsymbol{\theta}(\frac{n^2}{\mathrm{p}} + \mathrm{n} + \mathrm{p})$, depending upon which way the all-to-all exchange is implemented.

3.4.5 Execution Plan

The execution for the parallel matrix-vector multiplication (column-wise decomposition) algorithm is as described in the case of the parallel matrix-vector algorithm in section

3.4.6 Results of Parallel Matrix-Vector Multiplication Algorithm (Column-wise Decomposition) on Wired and Wireless LAN

Also in this experiment, a 4000 x 4000 matrix whose elements are of data type double was multiplied with a 4000 x 1 vector whose elements are also of data type double. A total of 100 runs were made. Just like in the row-wise decomposition, after each set of cluster runs, the average execution time is obtained. The experiment was also carried out on a maximum number of 10 clusters. Figure 3.10 show graph of the results obtained

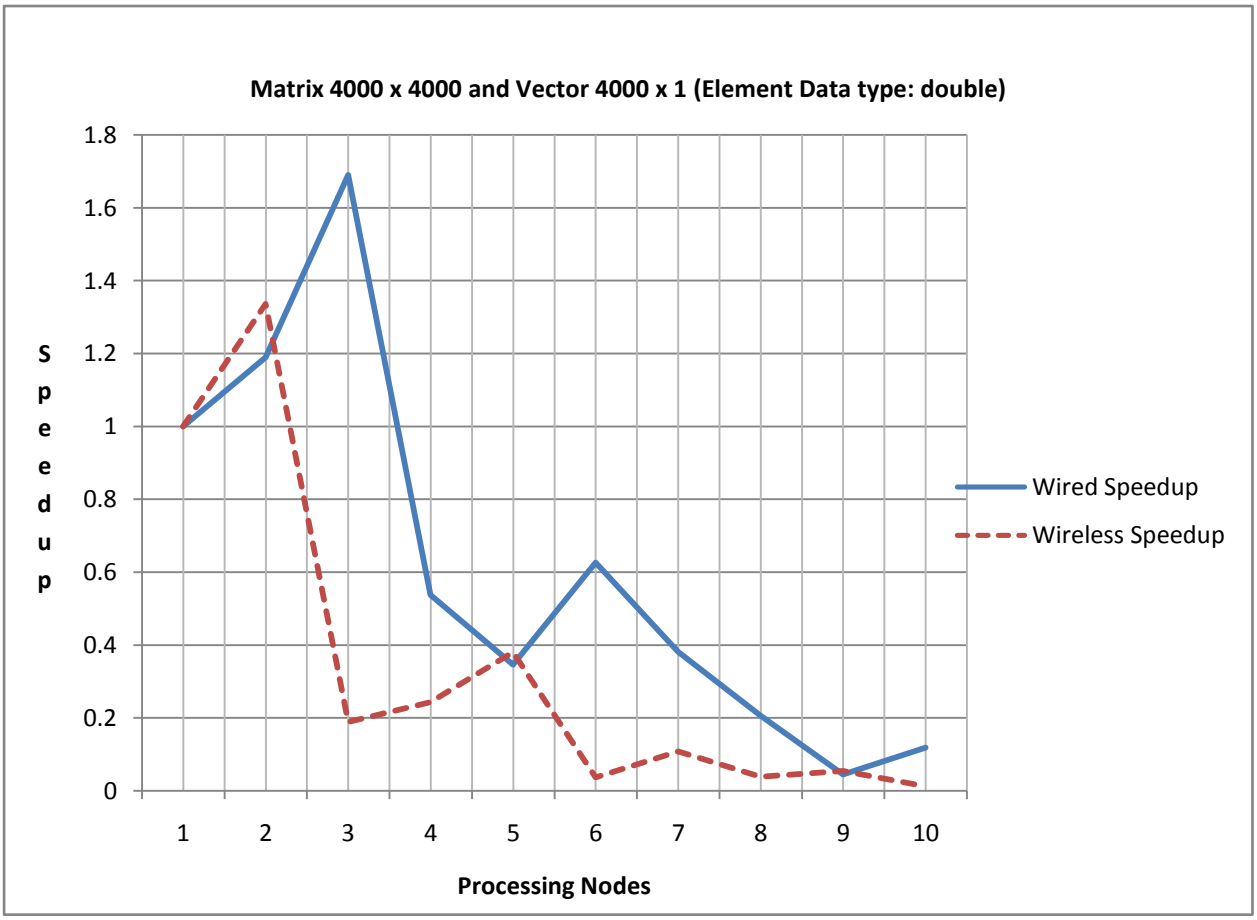


Figure 3.10: Performance for parallel matrix-vector multiplication using column-wise decomposition

Just like the row-wise parallel matrix-vector multiplication algorithm, a similar result is obtained. In this graph, it is also observed that after the execution on two clusters, performance degrades for the wireless setup. At this point, there are three nodes contending for the wireless media, hence there is likely to be contention of resources. The nature of the wired connection makes it impossible for collision since a switch was used to setup the network and a typical Ethernet switch breaks-up collision domains among its interfaces.

The Jumpshot views in Figures 3.11 to Figures 3.14 of the executed program give further details as to why there is degradation in performance on the wireless network. Just as explained in the case of the parallel matrix-vector algorithm (row-wise decomposition), the same delay in the send and receive primitives of the MPI communication is experience in the wireless LAN in Figure 3.13 compared to Figure 3.11 which is the wired equivalent of the same program.

Also, as the number of nodes increase, the average send/receive time of the wireless LAN increases compared to that of the wired LAN that decreases. This clearly shows contention of media resources in the wireless LAN. Just as mentioned earlier, it is expected that as the number of nodes increases, so also will the chunk allocated to each processing node decrease therefore, if there is no contention or delay in the medium, the average amount of time needed to send/receive data chunk should decrease. This decrease in average/send receive is what is observed in the wired LAN and not the wireless LAN.

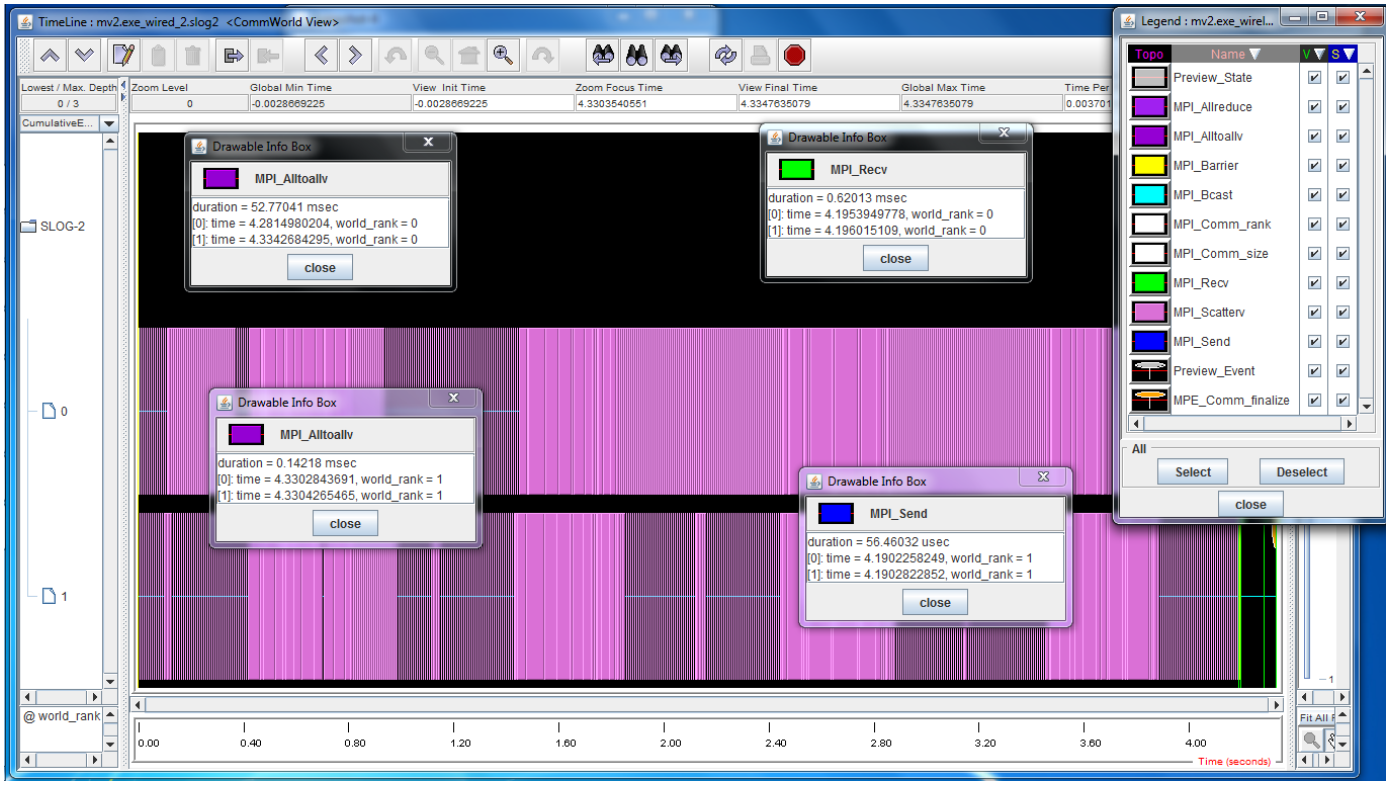


Figure 3.11: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 2 Cluster wired LAN

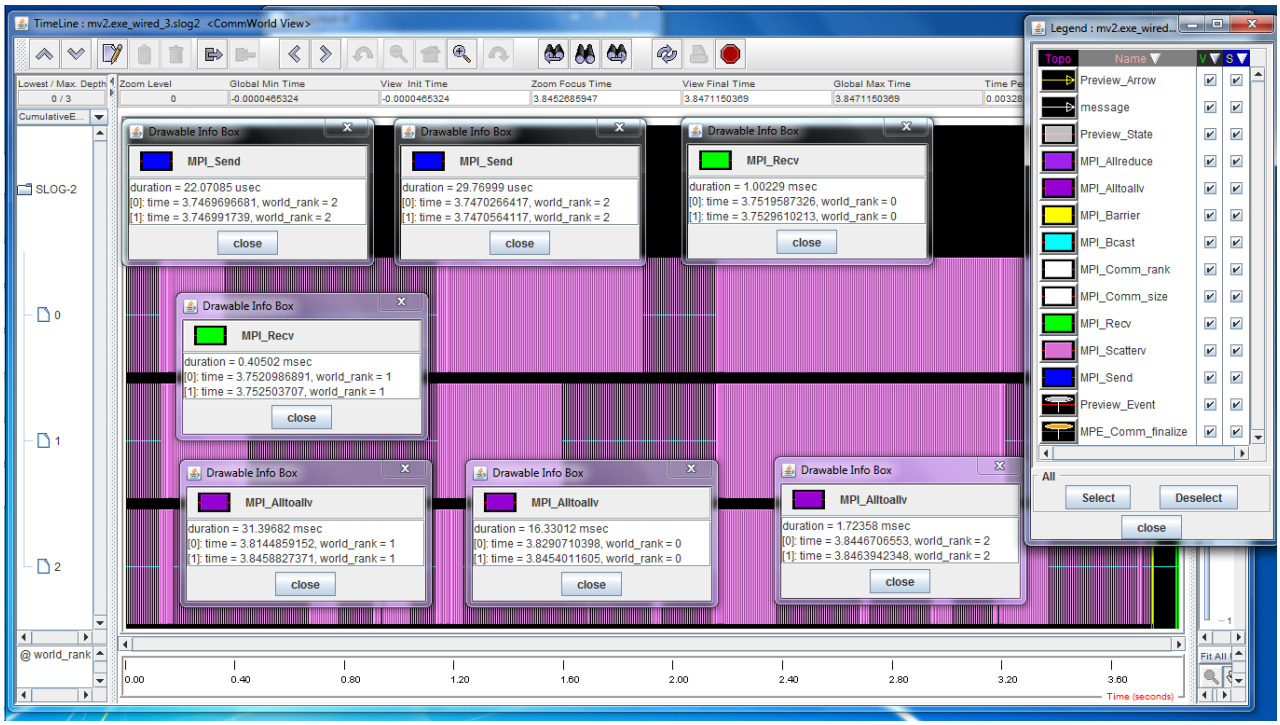


Figure 3.12: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 3 Cluster wired LAN

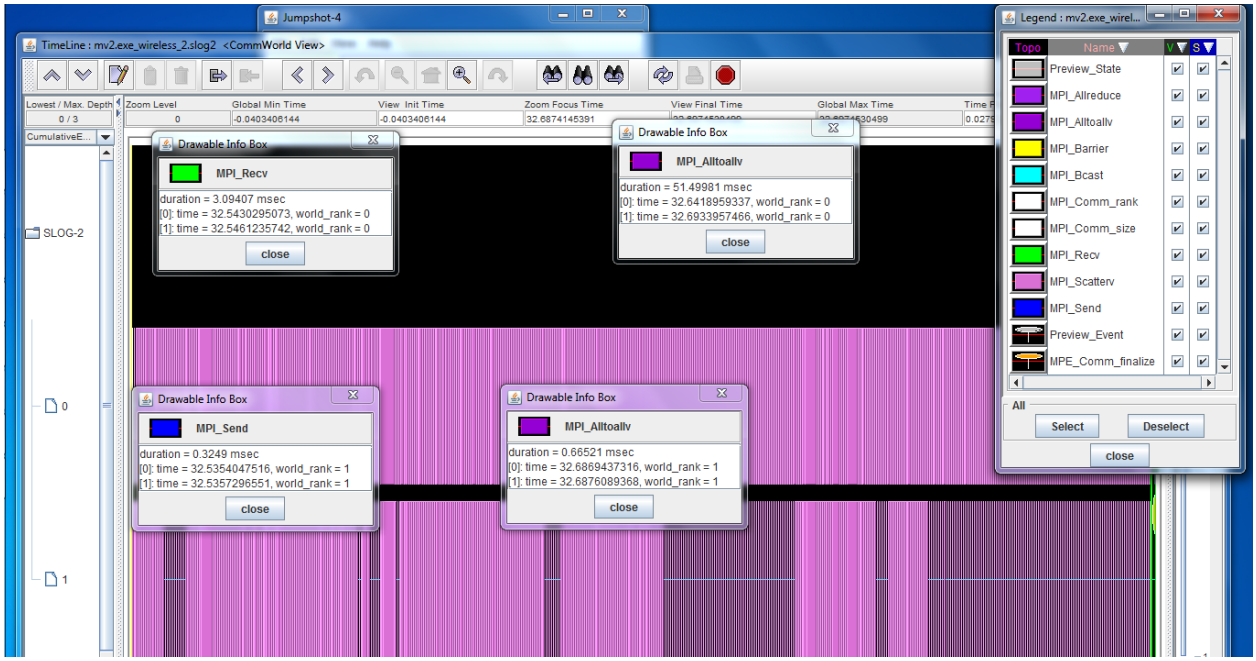


Figure 3.13: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 2 Cluster wireless LAN

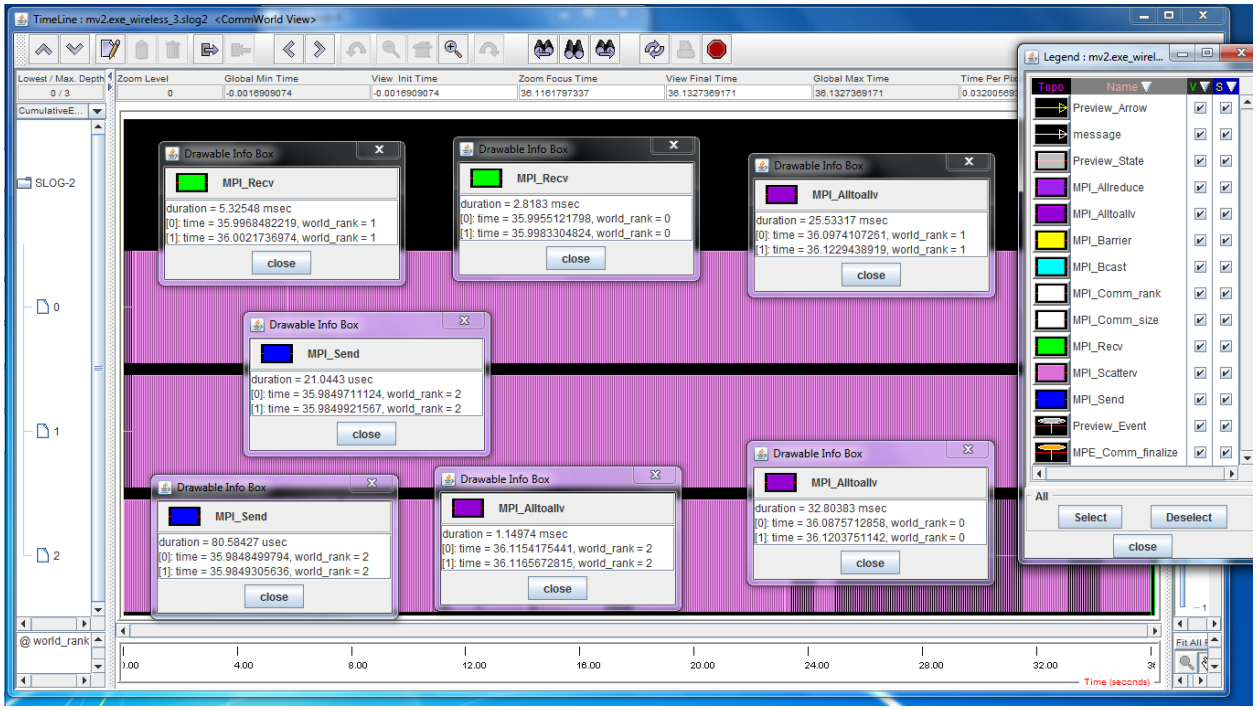


Figure 3.14: Jumpshot view of parallel matrix-vector multiplication (column-wise decomposition) on 3 Cluster wireless LAN

3.4.7 Parallel Matrix-Vector Multiplication Algorithm (Checkerboard Decomposition)

Unlike the row-based and column-based matrix decomposition, in checkerboard parallel multiplication algorithm, domain decomposition is done in a two dimensional grid-like form and each grid is associated with an MPI process. The matrix domain decomposition is depicted in Figure 3.15

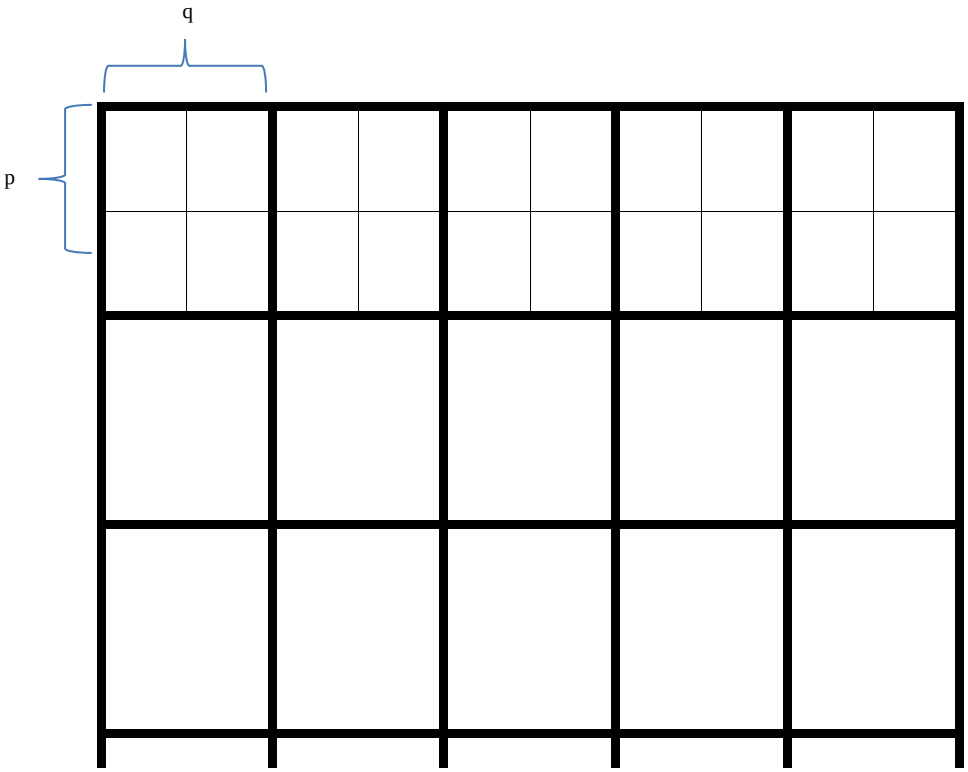


Figure 3.15: Checkerboard decomposition of parallel matrix-vector algorithm

Checkerboard Primitive Task

In the checkerboard parallel matrix-vector, the primitive task is to multiply$a_{i,j}$by b_j, to yield $d_{i,j}$. Each element c_i of the resultant vector is $\sum_{j=0}^{n-1} d_{i,j}$. In effect, for each row i, all the $d_{i,j}$ terms are added to produce element i of vector c, as illustrated below

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} => \begin{bmatrix} d_{0,0} & d_{0,1} & d_{0,2} \\ d_{1,0} & d_{1,1} & d_{1,2} \end{bmatrix} => \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

Agglomeration and Mapping

Agglomeration and Mapping step involves combining the primitive tasks into rectangular blocks and associate each task with a rectangular block to an MPI process as shown in Figure 3.15. And the process in the first column gets portions of vector b as illustrated in Figure 3.16

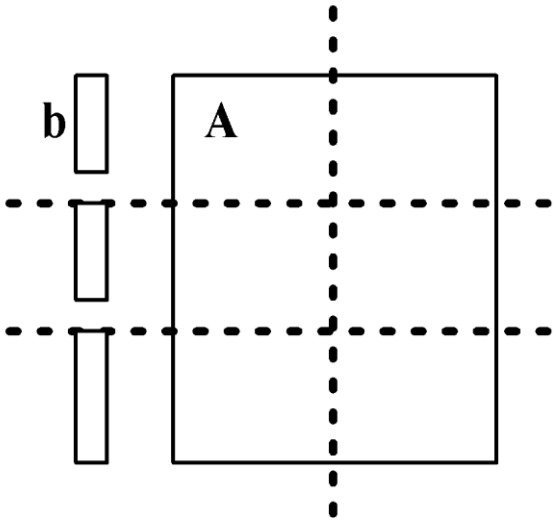


Figure 3.16: Partitioning of Vector B and Matrix A

Communication

After agglomeration phase, there are three principal steps that form the checkerboard matrix-vector multiplication algorithm

1. Each process or task has to be allocated the right portion of vector B for its computation. Hence, portions of vector bhave to be decomposed and redistributed among the processes in the communication grid as depicted in Figure 3.16
2. Each process performs a matrix-vector multiplication with its portions of A and B.
3. Processes in each row perform a sum-reduction on their portions of c. After the sum-reduction, result vector c is distributed by blocks among the tasks in the first column of the task grid. These steps are depicted in Figure 3.17

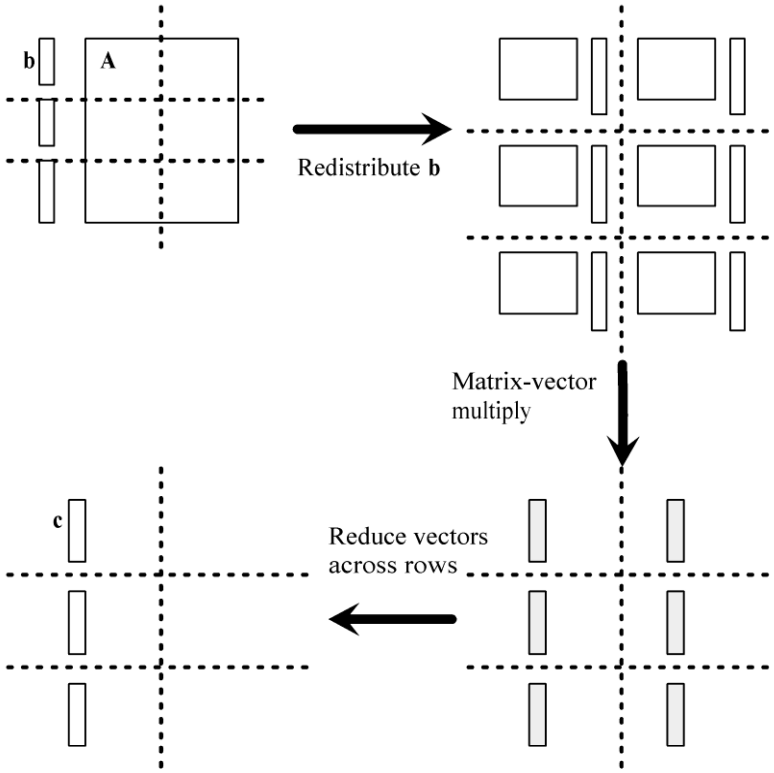


Figure 3.17: Parrallel Matrix-vector multiplication steps (checkerboard decomposition)

The Distribution of vector b among the participating processes is nontrivial given that the number of partaking processes may not at all time be a square. Assuming that the processes partaking in the computation are divided into a k x l grid, then after redistribution, a copy of b is divided among the processes in each row of the process grid.

Also, the number of processes in the computation/communication grid may not always be square, that is $k = l$ is not true at all execution cases. This gives rises to two conditions and separate computational steps for both cases.

In general, the redistribution of vector b involves two steps which is as follows

Step One

Move vector bfrom processes in first column to processes in first row. In doing this, the case of whether $k = l$ will be considered

i. If $k = l$, that is, number of processes is a square, then first column/first row processes send/receive portions of b

ii. If $k \neq l$, that is, number of processes is **NOT** a square,

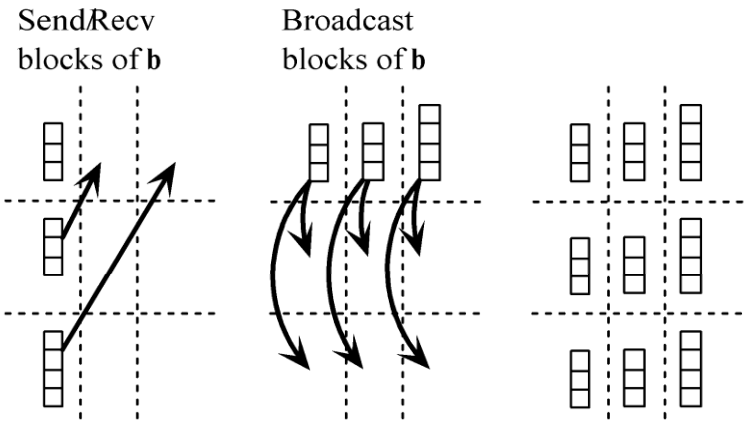
   a. thenGather vector b on process on grid location 0,0

   b. Process on grid location 0,0 broadcasts to first row processes

Step Two

First row processes scatter b within columns

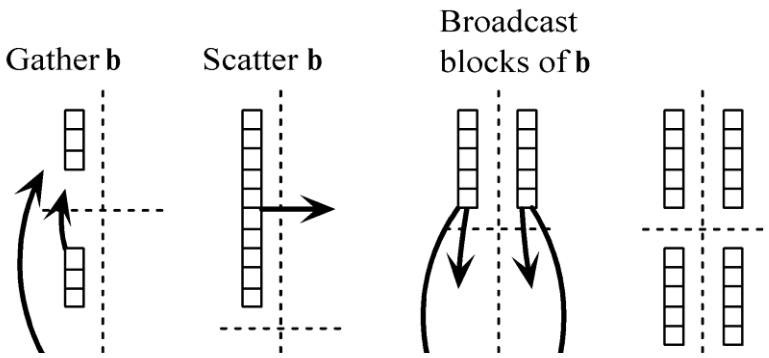
These two steps are depicted in the Figure 3.18 and Figure3.19



When k =l, i.e number of processes is a square

(a)

Figure 3.18: Parrallel Matrix-vector multiplication steps (checkerboard decomposition)



When $k! = l$, i.e number of processes is not a square

Figure 3.19: Showing parallel checkerboard matrix-vector multiplication algorithm when number of process is NOT a square

(b)

Analysis

Assuming$m = n$ and p is a square(best case analysis), then each process is responsible for not more than $\lceil n/\sqrt{p} \rceil$ x $\lceil n/\sqrt{p} \rceil$, therefore the time complexity is $\boldsymbol{\theta}(\frac{n^2}{p})$.

When p is a square, the redistribution of vector b is done in two steps. First each process in the first column sends its portion of vector b to the process in the first row. The time required for this message-passing step is $\boldsymbol{\theta}(\frac{n}{\sqrt{p}})$.

Next, each process in the first row broadcasts its portion of vector b to the other processes in the same column. This broadcast has time complexity$\boldsymbol{\theta}(\log\sqrt{p}\,(\frac{n}{\sqrt{p}})) = \boldsymbol{\theta}(\frac{n\log p}{\sqrt{p}})$.

After the matrix-vector multiplication step, processes in each row sum-reduce their portions of B. The time needed for this communication operation is O(log $\boldsymbol{\theta}(\log\sqrt{p}\,(\frac{n}{\sqrt{p}})) = \boldsymbol{\theta}\left(\frac{n\log p}{\sqrt{p}}\right)$

Combining these terms, the overall time complexity of the parallel matrix vector multiplication algorithm based on a checkerboard block decomposition of matrix is $\boldsymbol{\theta}(\frac{n^2}{p} + n\frac{\log p}{\sqrt{p}})$.

3.4.8 **Execution Plan**

The execution for the parallel matrix-vector multiplication (checkerboard decomposition) algorithm is as described in the case of the parallel matrix-vector algorithm in section 3.3.2.

3.4.9 Results of Parallel Matrix-Vector Multiplication Algorithm (Checkerboard Decomposition) on Wired and Wireless LAN

The last parallel matrix-vector algorithm looked at is the algorithm involving the checkerboard decomposition. This is because its layout and algorithm behaviour is very different from the previous two algorithms observed. In this algorithm, a 4000 x 4000 matrix whose elements are of data type double was multiplied with a 4000 x 1 vector whose elements are also of data type double. Also a total of 100 runs were made with ten runs each for the various numbers of clusters observed. Figure 3.20 shows the result of the executions

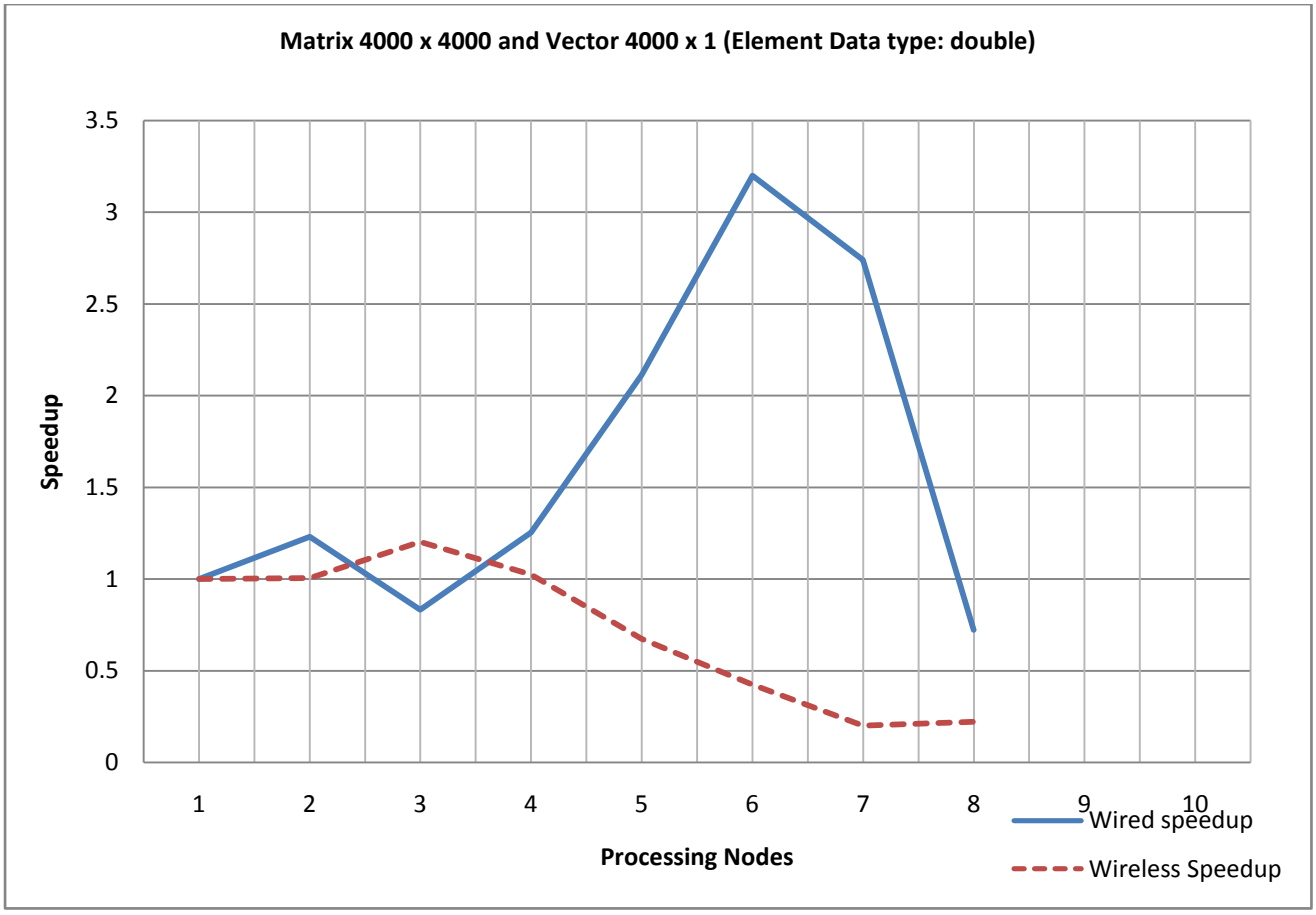


Figure 3.20: Performance for parallel matrix-vector multiplication using checkerboard decomposition

Just like the previous two algorithms analysed, the same degradation in wireless LAN performance is observed for the algorithm involving the checkerboard decomposition. It should be also observed that since computation overhead is much less compared to communication overhead in relation to the speed of computation for each cluster (as seen from the running time of the algorithm compared with the speed of each node), the effect of communication on performance still contributes immensely to the sharp drop on performance of the algorithm.

However, there is a performance increase at 3 clusters for wireless LAN and a performance decrease at 2 clusters on wired LAN. This is a slight deviation from the behaviour of the previously observed algorithms and it is largely due to the partitioning

and arrangement of the processes forming the computation/computation grid. At three clusters the processes form a 3 x 1 grid reducing the algorithm to a simple row-wise decomposition with addition network overhead for MPI_Gather, MPI_Scatter and MPI_Broadcast operations. The send/receive for both average time still exhibits a decrease on wired and an increase on wireless as seen in figures 3.21 and 3.22 but in this special case, MPI_Broadcast dominates most part of the algorithm process and wireless LAN tend to do better since it is a broadcast medium.

Compared to other algorithms like the parallel matrix-vector multiplication (row-wise decomposition) and parallel matrix matrix-vector multiplication (column-wise decomposition), the parallel matrix-vector multiplication algorithm(checkerboard decomposition) slows the degradation in performance on the wireless network. This is because broadcast consists most part of the algorithm.

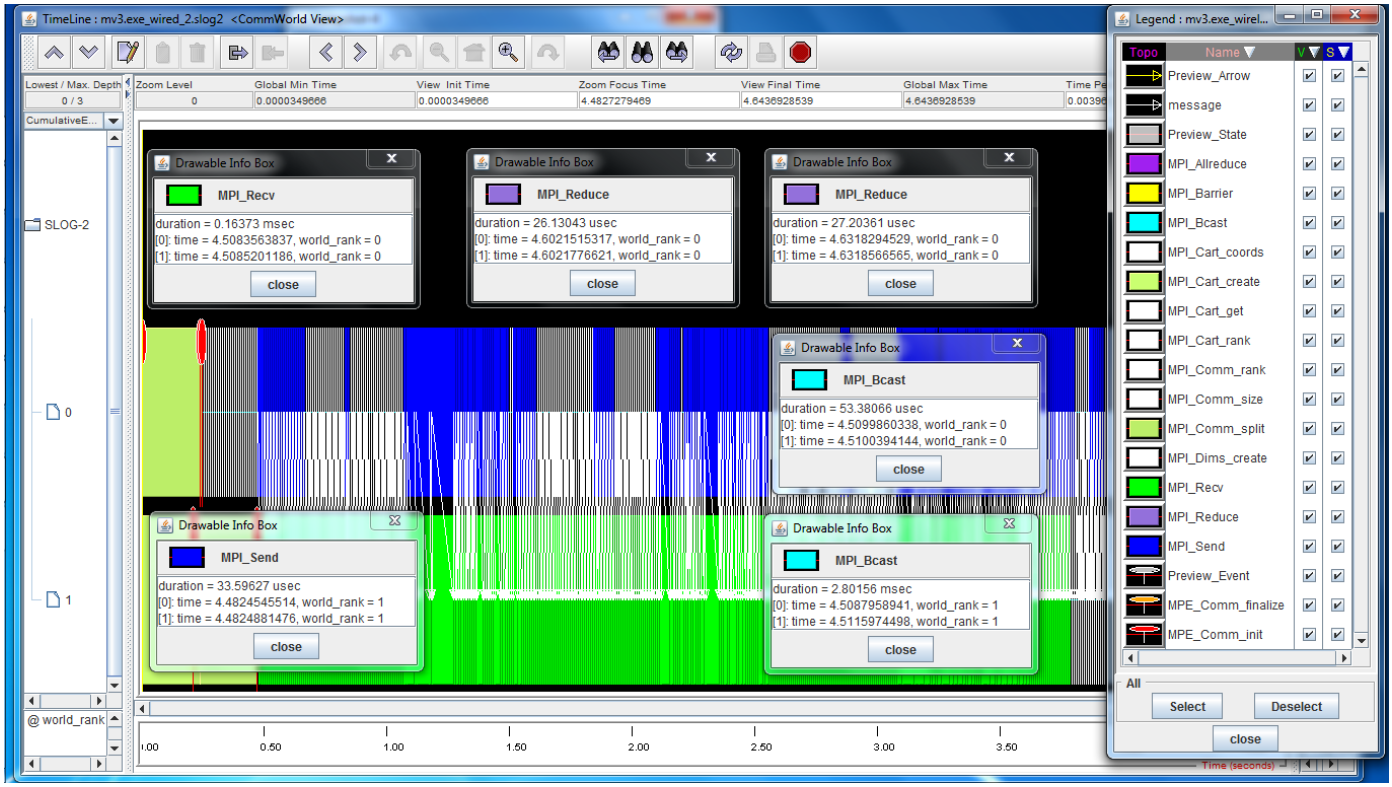


Figure 3.21: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 2 Cluster wired LAN

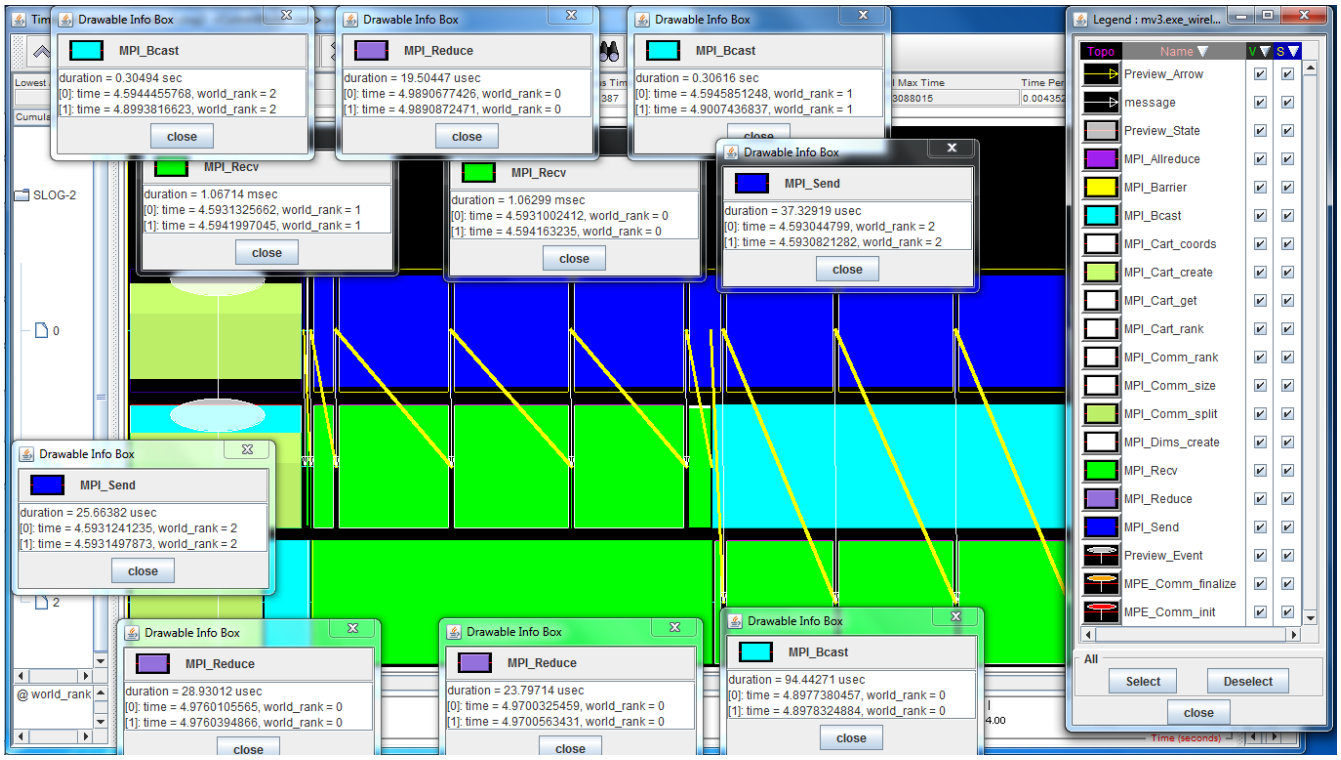


Figure 3.22: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 3 Cluster wired LAN

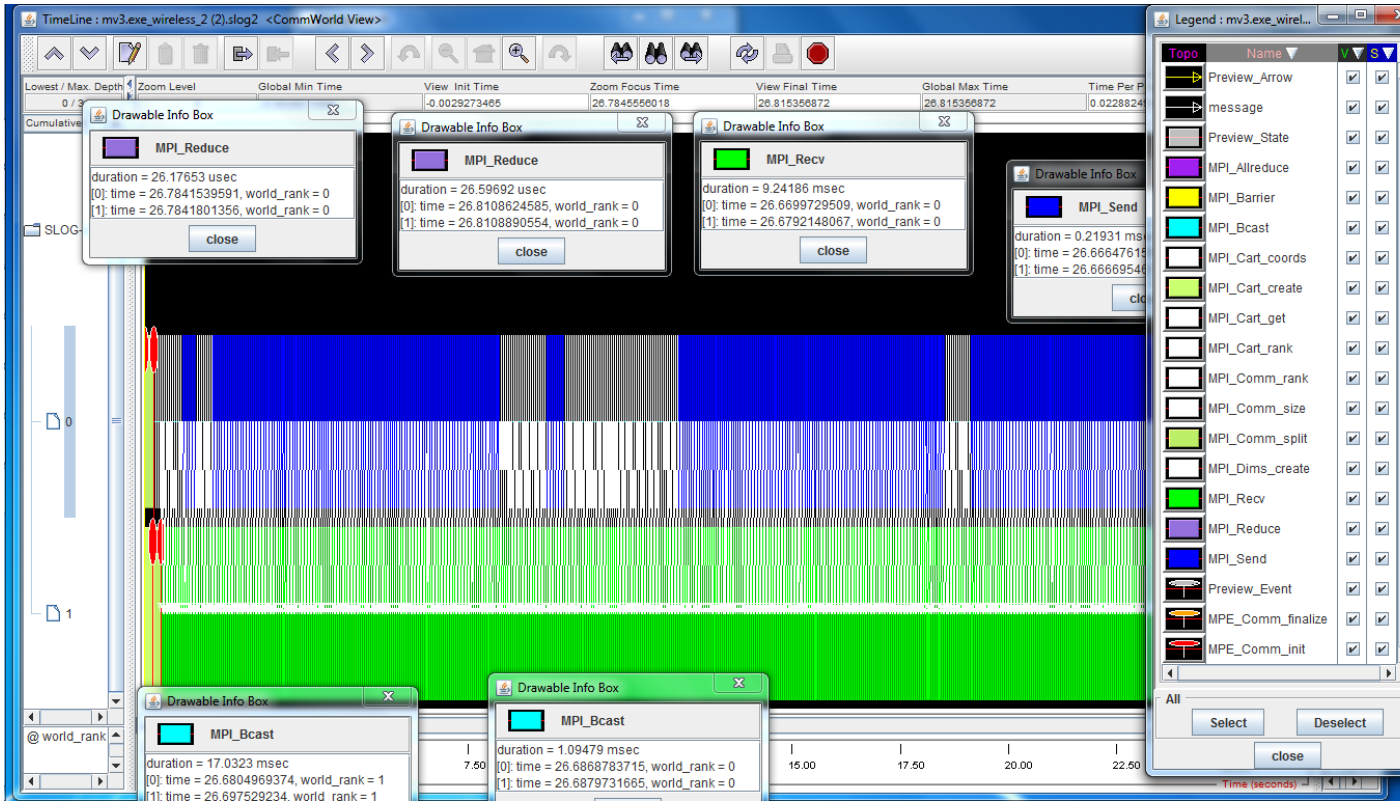


Figure 3.23: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 2 Cluster wireless LAN

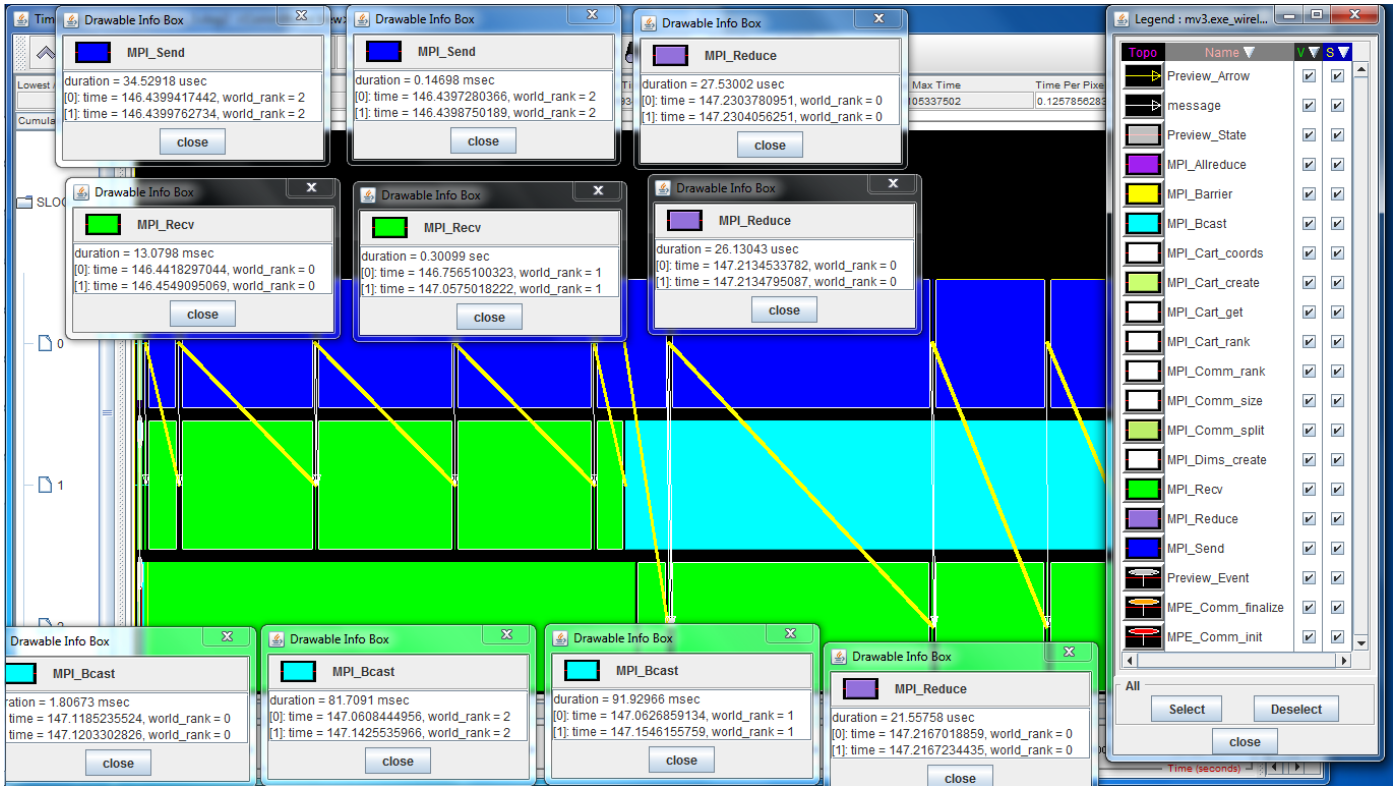


Figure 3.24: Jumpshot view of parallel matrix-vector multiplication (checkerboard decomposition) on 3 Cluster wireless LAN

3.5 Chapter Summary

In this chapter, a detailed description of the experiment setup was done. Also described were the various parallel matrix-vector algorithms(row-wise, column-wise and checkerboard decomposition) used for this analysis. The description for the algorithms included the decomposition techniques, algorithm design and algorithm time complexity analysis of the performance of these algorithms. The execution plan was described followed by the results obtained after each parallel algorithm has been executed on both wired and wireless LAN.

In general, there is degradation in performance on wireless LAN as the number of nodes in the cluster increases. This is in contrast to the wired LAN behaviour. However, there is a limit to the performance gain on wired LAN. This limit is attained when the communication cost outweighs the computation cost.Also, irrespective of the network, performance gain achieved through modification of parallel algorithm is seen on both wired and wireless networks.

CHAPTER FOUR

PARALLEL MATRIX-MATRIX MULTIPLICATION EXPERIMENT

4.1 Introduction

This chapter contains details of how the parallel matrix-matrix multiplication experiment for this empirical analysis was carried out. It contains a brief explanation of the sequential matrix-matrix multiplication algorithm followed by the parallelized matrix-matrix multiplication algorithm using row-wise decomposition.

This chapter also discusses the experimental results and the degradation observed in the wireless LAN. Also, it introduces the multi-programming approach which helps improve the performance of the parallel program on wired and wireless LAN.

4.2 Experimental Setup

The experimental setup which includes the system configuration, network setup, operating system, parallel library setup, program compilation and execution plan is as discussed in chapter three of this thesis.

4.3 Sequential Matrix-Matrix Multiplication Algorithm

Given a matrix $A$and B with dimensions A_{mn} and B_{nm} where m and n are integers ranging from 0 to $m-1$, $n-1$ respectively, the product matrix C_{mn} is defined as the product of $A_{mn}.B_{nm}$. Therefore, $C[c_{ij}] = \text{A.B}$, $c_{ij} = \sum_k^n A_{ik}.B_{kj}$ where c_{ij} is an element of product matrix C and k are values ranging from 0 to $n-1$. The sequential naïve implementation of this algorithm requires three nested loops with the inner loop calculating the c_{ij} term of the resultant matrix$C$. Therefore, the running time for this

algorithm is $\boldsymbol{\theta}(mn^2)$. In the case of a square matrix A with the same dimension as matrix B, then the running time for the sequential naïve algorithm will be $\boldsymbol{\theta}(n^3)$.

4.4 Parallel Matrix-Matrix Multiplication Algorithm (Row-wise decomposition)

Sequential Matrix-Matrix multiplication algorithm has a time complexity of $\boldsymbol{\theta}(n^3)$. Because of this high computation overhead, it lends itself as a very good algorithm to be parallelized.

In this parallel matrix-matrix multiplication algorithm, domain decomposition is row-wise. Each matrix A and B is decomposed (just like the row-wise matrix decomposition in the row-wise matrix-vector multiplication algorithm) and distributed among the processes partaking in the computation.

The steps in the parallel matrix-matrix multiplication algorithm is as follows

4.4.1 Decomposition and Primitive Task

In this algorithm, the primitive task involves multiplying a row from Matrix A with a row from Matrix B producing a partial resultant sub-matrix C. Multiplication is done such that given a row i of matrix A and a row i of Matrix B, each computation produces $c_{i,1}$, $c_{i,2}$, $\ldots c_{i,n}$ of the partial resultant sub-matrix C, where n is the number of columns of Matrix B. Thus elements $c_{i,j}$ of Matrix C form the partial resultant sub-matrix of C.

This is illustrated in Figure 4.1

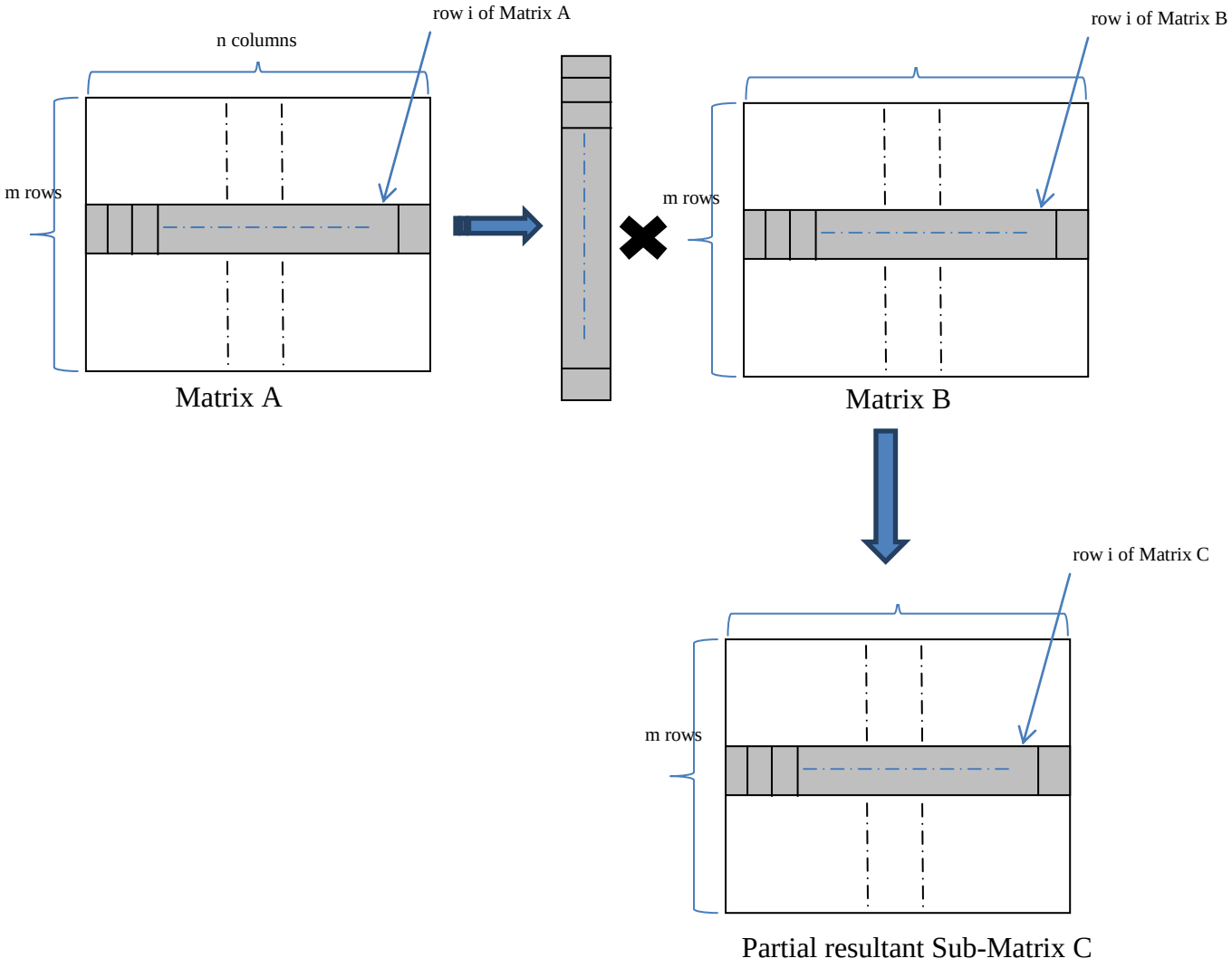


Figure 4.1: Showing the primitive task involved in parallel matrix-matrix multiplication with row-wise decomposition

4.4.2 Agglomeration and Mapping

In matrix-matrix multiplication, the resultant matrix C is calculated without modifying the Matrix $A$and B, hence each Matrix A and B are decomposed row-wise and distributed across the processes in the computation grid.

Just like the row-wise matrix-vector parallel algorithm, each process is assigned not more than $\lceil n/p \rceil$ of the Matrix A and of the Matrix B.

Therefore, each process computes its portion of its allocation by multiplying its sub-matrix A and sub-matrix$B$ producing a partial resultant sub-matrix C. With row i of matrices of $A$and B, each process y can compute $a_{i,j}.\mathrm{b}_{i,0}$, which is one of the terms of $c_{i,0}$. It can also compute $a_{i,i}b_{i,1}$ which is one of the terms of $c_{i,1}$ and so on. In order words, the process can perform n multiplications that represent partial sums for the n elements of row i of resultant Matric C where n is the number of columns of B.

This computation can be further view in figure 4.2 below

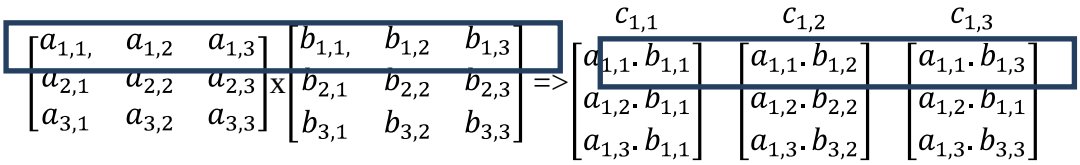


Figure 4.2: Partial multiplication of Matrix A and B producing a partial Resultant Matrix C

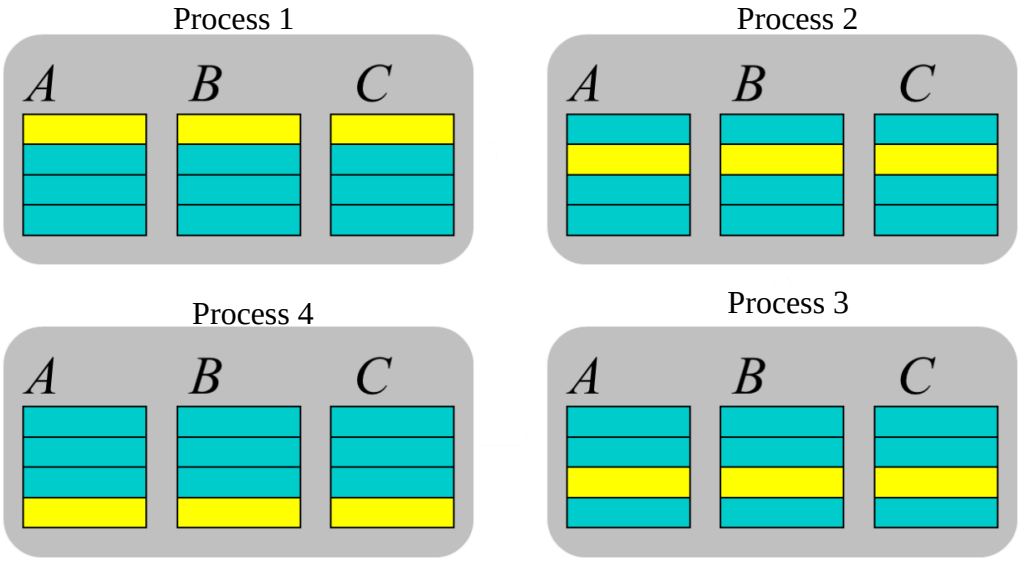
4.4.3 Communication

The computation under the agglomeration phase only produces a partial resultant matric C and at the end of this phase, each process i will have a partial resultant matrix.

Each process then needs rows from all other $p-1$ process in the grid to compute the final sub-matrix for the resultant matrix C, where p is the number of processes in the communication grid.

One way to exchange rows among all the process in the communication grid is to perform a **ring-like** exchange of rows where process i sends its rows to process $i + 1$ and receives rows from process $i - 1$.

The ring-like exchange of portions of matrix B among processes is as depicted in Figures 4.3, 4.4, 4.5 and 4.6



4.3: First phase of the ring- exchange of allotted portion of B

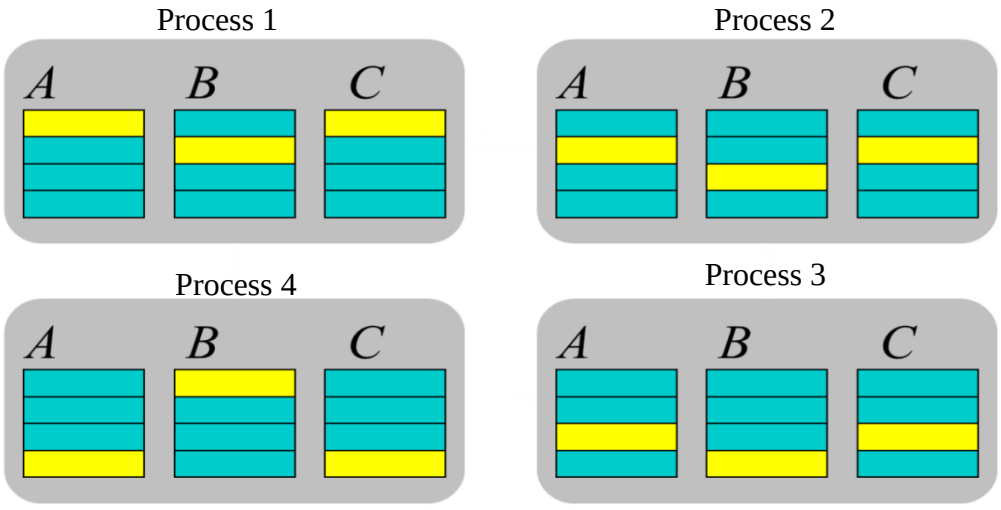


Figure 4.4: Second phase of the ring-exchange of allotted portion of B.

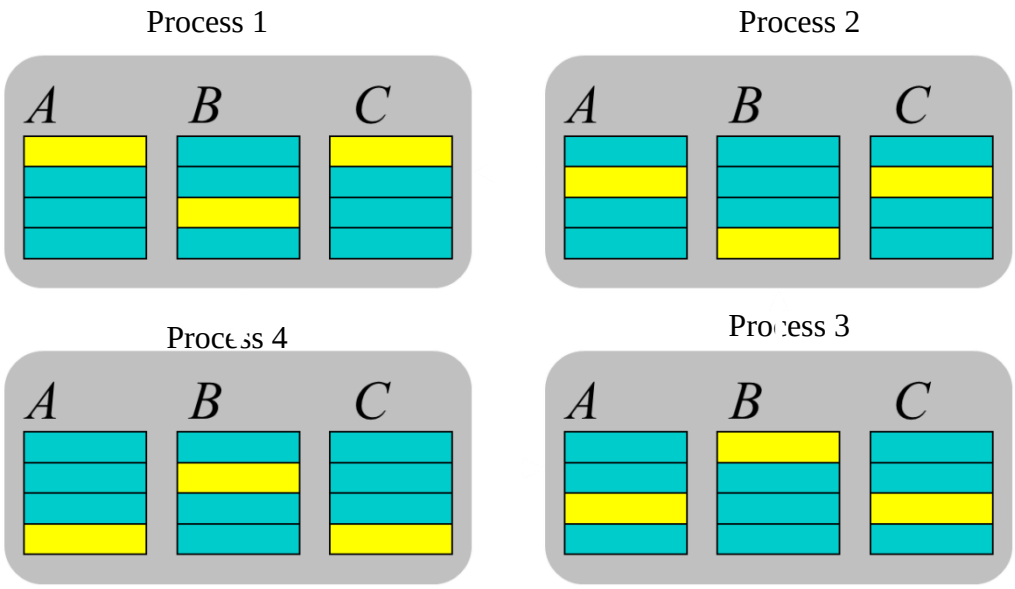


Figure 4.5: Third phase of the ring-like exchange of the allotted portion of B.

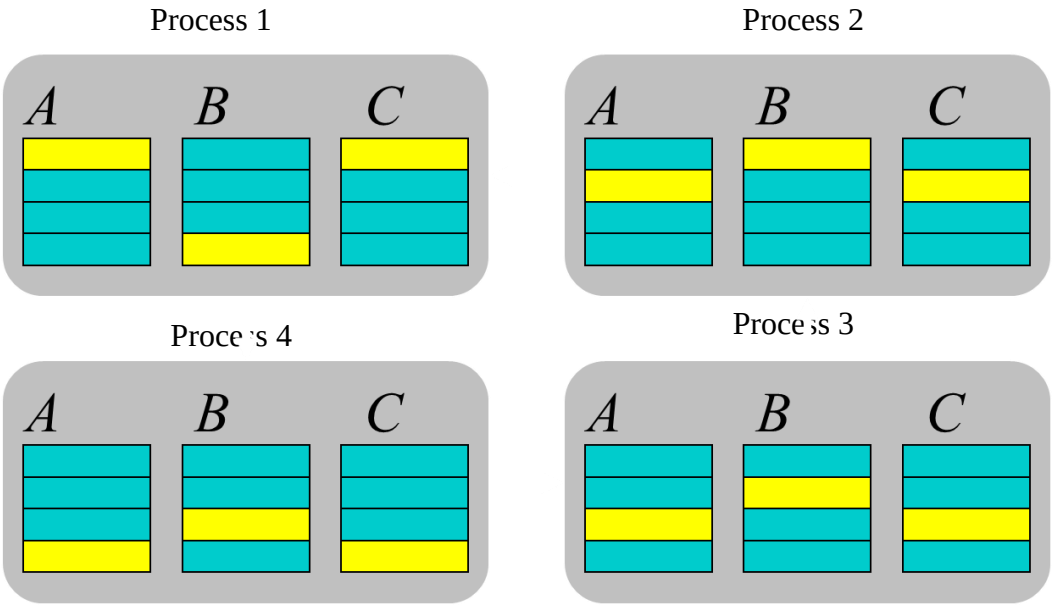


Figure 4.6: Final phase of the ring-like exchange of the allotted portion of B.

Figures 4.3 to 4.6 show the exchange of rows among processes in the communication grid. In these Figures it is assumed that there are four processes process 1, 2, 3 and 4. Each assigned a corresponding row in the matrices A, B and C.

Figure 4.3 shows this initial allocation and the first phase. The yellow rows indicate the rows allocated to each process. After computation, each process i sends its copy of row of Matric B to process $i + 1$ and receives a copy of row B from process $i - 1$.

This communication pattern continues in Figure 4.4, 4.5 and 4.6. At the final phase, each process would have received $i - 1$ messages and would have sent $i - 1$ messages, therefore having enough rows for multiplication.

In order to avoid deadlock, this communication process was performed by first allowing all the odd numbered process to receive rows before sending their various rows and all even-numbered processes sends their rows to odd-numbered processes before waiting to receive from these odd-numbered processes.

At the end of the exchange of allotted portions of Matrix $A$and B, each process now has a partial matrix$C$ of the resultant matrix C. An all-gather is then performed to bring together these partial sub-matrices to form the resultant Matric C

4.4.4 Analysis

Assuming that A, B, and C are all $n\ x\ n$ matrices. Assuming also that n is a multiple of p where p is the total number of processes. Each process in the computation grid controls the same n/p rows of A and C throughout the algorithm. The contiguous groups of n/p rows of B are passed from process to process.

When the algorithm begins, each process initializes its $(n/p)\ x\ n$ portion of $C$to 0. During each iteration every process multiplies an$(n/p)\ x\ n$ block of A by the

$(n/p)\ x\ n$ portion of B it currently possesses. It then adds up the resultant $n/p\ x\ n$ to its portion of C.

If γ is the time neeed for one of the add-multiply steps inside the inner product, then the computational time of each iteration is $\gamma\left(\frac{n}{p}\right)\left(\frac{n}{p}\right)n = \gamma n^3/p^2$

During every iteration, each process must also communicate its portion of B to the next process on the ring. If the communication is done after computation, the time needed to send these elements would add $\alpha + \frac{\left(\frac{n}{p}\right)n}{\beta}$ to the execution time of each iteration. Where α is the unit of time to send the message and β is the bandwidth of the channel.

Receiving the next section of B from the predecessor ring process would occur at the same time.

The algorithm has p iterations, therefore, the total computation and communication time, then is$\boldsymbol{O}(p\left[\frac{\gamma n^3}{p^2} + \alpha + \frac{n^2}{p\beta}\right] = \frac{\gamma n^3}{p} + p\alpha + \frac{n^2}{\beta})$

4.5 Execution Plan

Just like the previous algorithms in chapter three, the parallel matrix-matrix algorithm follows the same principle of execution. There is a separate program that generates one file per matrix and the elements of these matrices are generated randomly and stored as 8 byte of type double in the matrix. According to the matrix generating program, the first line of the matrix file contains the dimension of the matrix, while subsequent lines are the randomly generated matrix elements. Since the matrix file is stored as bytes as opposed to plan text, it is hardly readable from a text editor. The source program for the matrix generator is in appendix A.

A separate program was also developed for generating the vector file. It follows the same principle as the matrix generating program where 8 bytes of randomly generated double type numbers are stored as bytes in the file and the first of the file contains the matrix dimension.

All algorithms, were designed in such a way that process 0 is responsible for reading matric/vector elements from file and distributing the elements using row-wise, column-wise or checkerboard decomposition technique.

It is worth noting that the first three algorithms mentioned in chapter three, (parallel matrix-vector multiplication using row-wise decomposition, matrix-vector multiplication using column-wise decomposition and matrix-vector multiplication using checkerboard decomposition) were presented(Michael, 2004); while parallel matrix-matrix multiplication algorithm was contributed by the present thesis with descriptions and utility functions adapted from (Michael, 2004).

All source codesfor these algorithms are provided in the appendix.

4.6 Results Parallel Matrix-Matrix Algorithm (row-wise decomposition) on Wired and Wireless LAN

Parallel matrix-matrix multiplication algorithm involves a lot of computation and as such makes a good algorithm for this comparative analysis. The sequential matrix-matrix algorithm runs at $\theta(n^3)$ with Strassen's algorithm running slightly better at$\theta(n^{2.81})$ (Thomus *et al* 2002). In this experiment, a 2000 x 200 matrix is multiplied b a 2000 x 2000 matrix. There are a total of 100 runs with 10 runs each number of clusters observed. The graph in figure 4.7 shows the outcome of the experiment

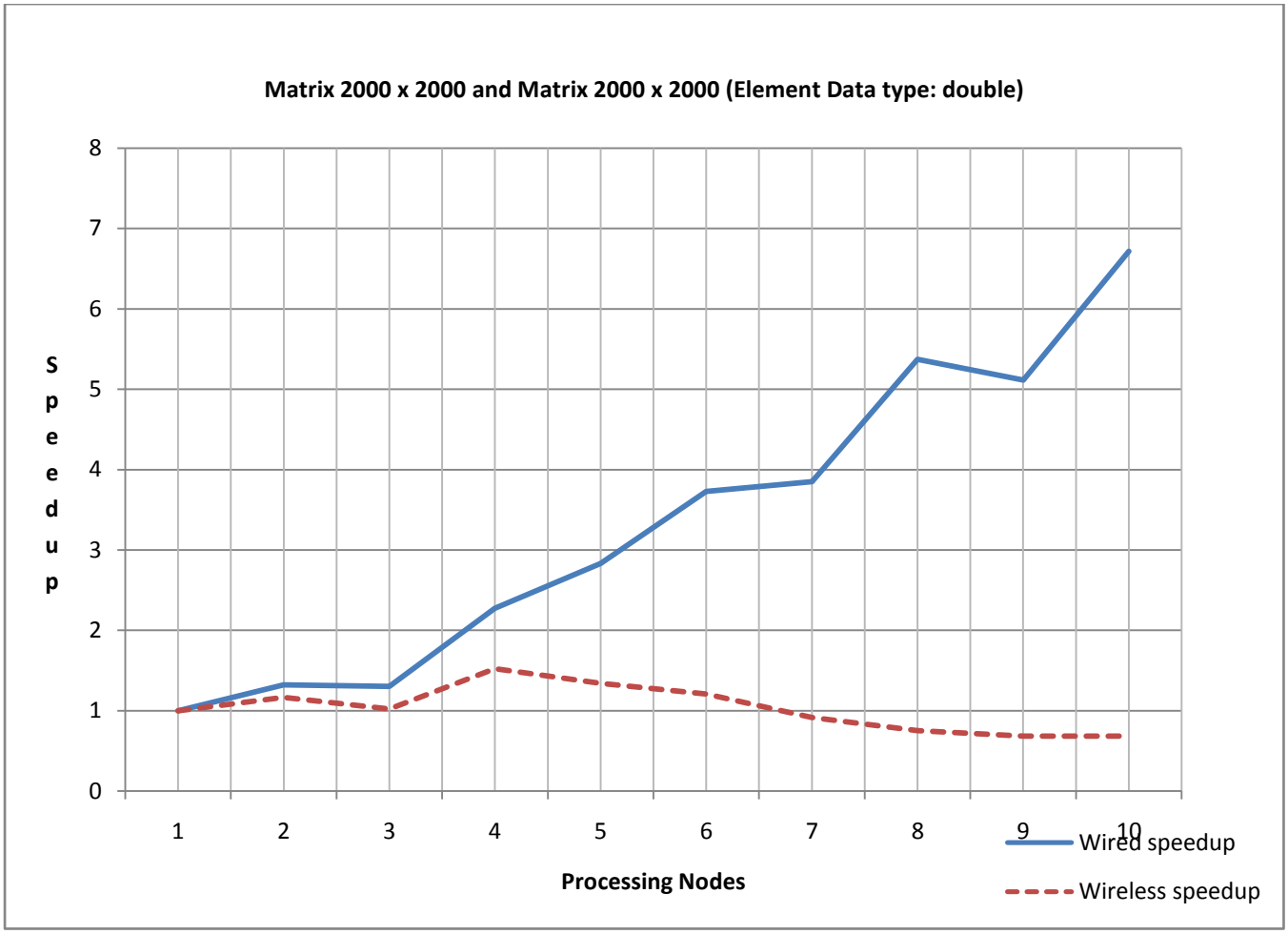


Figure 4.7: Performance graph for Parallel Matrix-Matrix Multiplication (row-base decomposition)

The graph in Figure 4.7also shows degradation as observed in the previous experiments. However, a slow degradation in the wireless LAN compared to the other algorithms observed. This is because of higher computational overhead in the parallel matrix-matrix algorithm. For every computation in the parallel matrix-matrix computation the time complexity is $\theta(n^3)$ compared to the matrix-vector algorithm that runs at $\theta(n^3)$ for each computation of chunk.

It shows therefore that the wireless setup has a negative impact on performance of parallel programs deployed under wireless networks. This degradation as explained in

the previous experiment is due to the nature of the wireless protocol and the possibility of collisions among data packets.

4.7 General Observation of the Algorithms

In all the algorithms discussed above, it should be noted that each computer in the cluster has a 2.2Ghz processor and 2GB of RAM with a 1024KB cache.Therefore for matrix-vector multiplication involving a 4000 x 4000 matrix and a 4000 x 1 vector and algorithm time complexity of $\theta(n^2)$, computation is appreciably faster on each individual system. The speed of the individual computer contributes immensely to the speedup observed for the parallel computation hence, the communication overhead forces a sharp decline in the overall performance.

Also, the nature of the Fast Ethernet LAN (Full Duplex) makes it possible to eliminate collisions among the nodes or computers partaking in the computation. Connectivity in Ethernet LAN was made possible via a switch device. A typical switch device is a layer 2 device that breaks up collision domain among its interfaces, therefore the possibility nodes sharing communication medium is eliminated thereby eliminating collision and in effect, there is no need for CSMA/CD protocol.

However, the same cannot be said of the wireless LAN (IEEE 802.11g) setup in this experiment. Connectivity is achieved through an access point thereby creating a Star topology with all the computers contending for the medium. Therefore, CSMA/CA's random backoff timer or the RTS/CTS overhead within the network increases the overall network overhead thereby quickly nullifying the effect parallel design would have had on the overall speedup. The speed of wireless LAN network also plays a negative role as it slowed down the overall speed of the algorithm because speed of communication also accounts for the general speed of the parallel algorithm. But the

greatest impact on the performance is the availability of collision thereby requiring the CSMA/CA protocol.

It is also worth noting that, there is delay in the nature of the send and receive primitives during communication. The four algorithms observed make use of a blocking send and receive mpi communication operation therefore, once a node is ready to send, the receiving node must be ready to receive data before actual data transmission takes place. Therefore, if a node delays in receiving data chunk, it will also affect its transmission if it also has to retransmit whatever it has received.

In the case of the matrix-matrix multiplication, it is already known from the serial algorithm that it is compute intensive, therefore there is a lot to speedup gain through parallelization of the algorithm. However, as observed from the performance of the algorithm on both wired and wireless clusters, the same degradation is experienced in the wireless cluster. This is because the scenario still remains the same. As more nodes are added to the computation, more contention is experience in the medium and the average delay in transmission/reception increases as the number of nodes increases and in turn the performance declines as the nodes increases.

The Jumpshot views shown in figures 4.8 to figure 4.13 also gave further evidences to the delays observed during send and receive primitives of the mpi communication. The dialogue boxes displayed to the left show the send and receive time at the beginning of the algorithm while the ones to the right show that send and receive time at the end of the algorithm.

Figures 4.8 and 4.11 show that the network contributes immensely to the general performance of programs and this can further be seen in Figures 4.9 and 4.12, Figures 4.10 and 4.13.

As the number of nodes increases from two nodes down to five nodes and eight nodes, the average send/receive operations for wired LAN decreases while that of the wireless LAN increases as observed with the three previous algorithms analysed. This is shown in Figures 4.8, 4.9, 4.10 and Figures 4.11, 4.12, 4.14.

Hence, it is clear from the jumpshot that the more the increase in the node the more the delay experience in the wireless LAN,hence the longer it takes to transmit and receive data between nodes, whereas in the case of the wired LAN, the more the nodes increases the faster it is to transmit data between nodes until the point where many nodes will mean too many communication overhead that surpasses the computation.

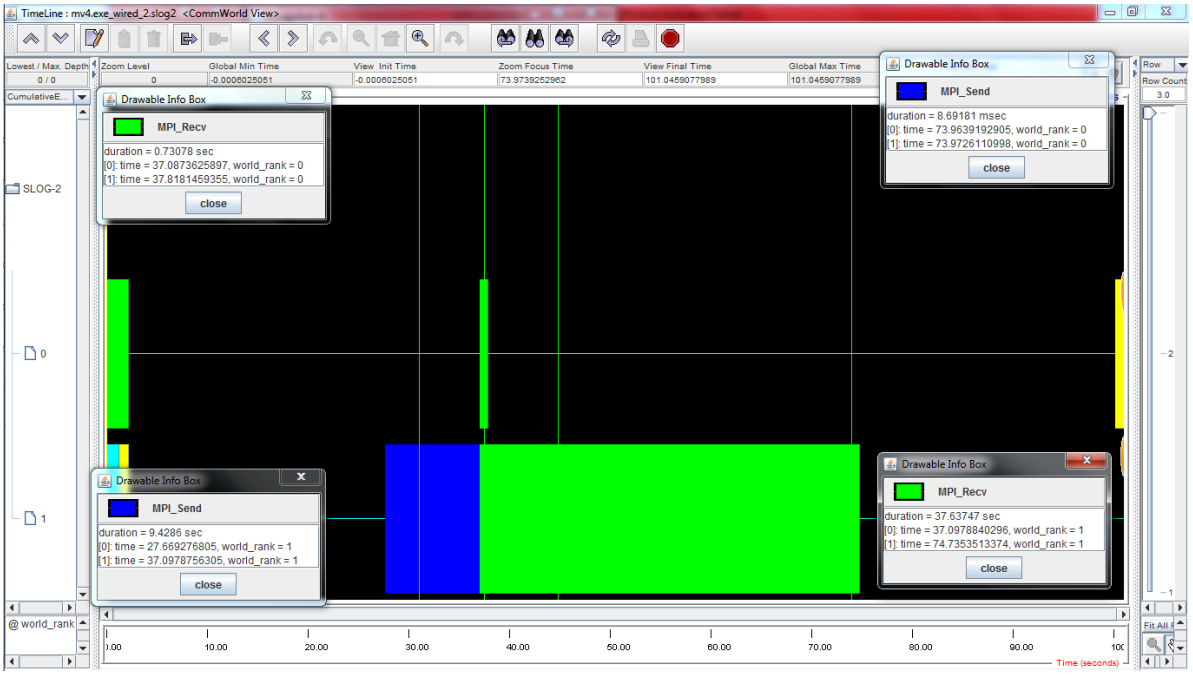


Figure 4.8:Jumpshot view of parallel matrix-matrix multiplication algorithm (row-wise decomposition) on 2 cluster wired LAN

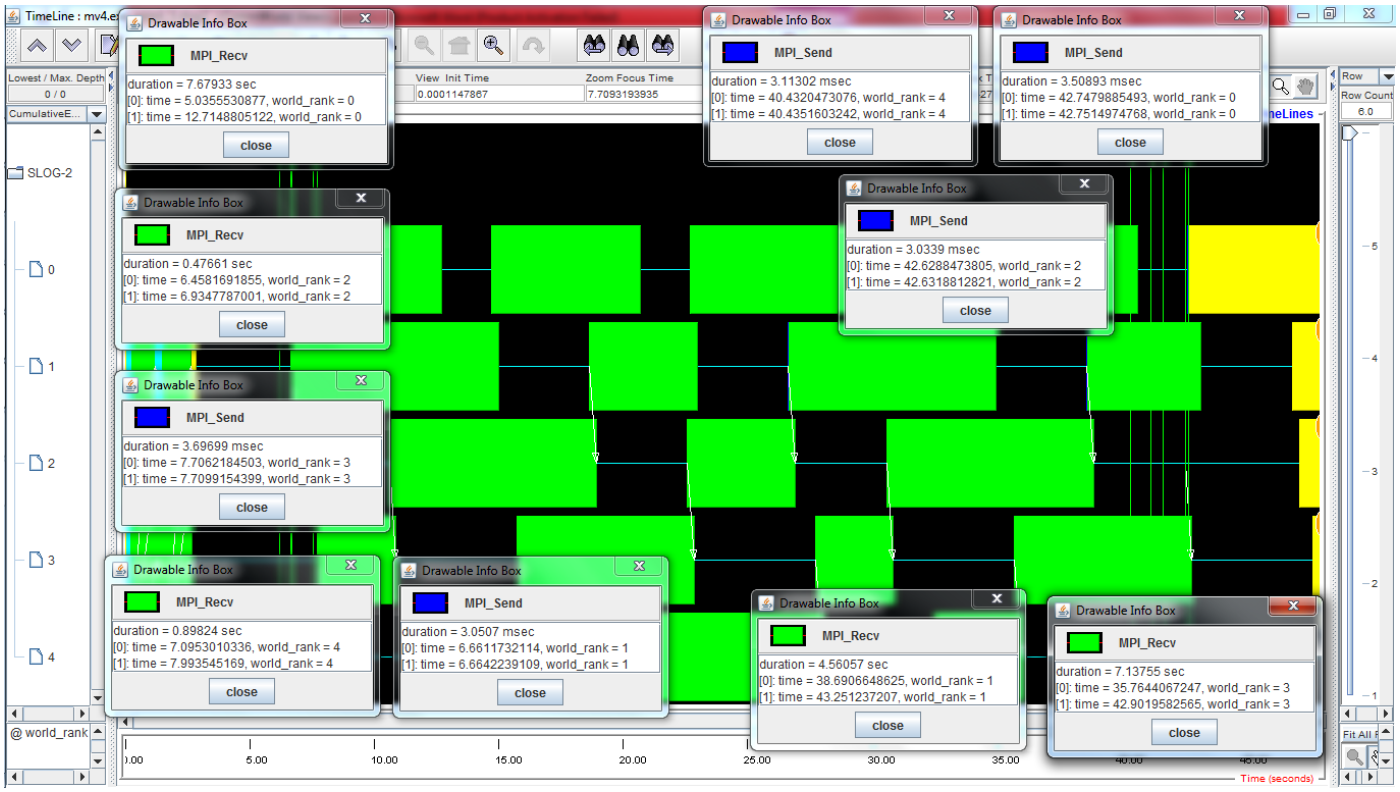


Figure 4.9: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 5 cluster wired LAN

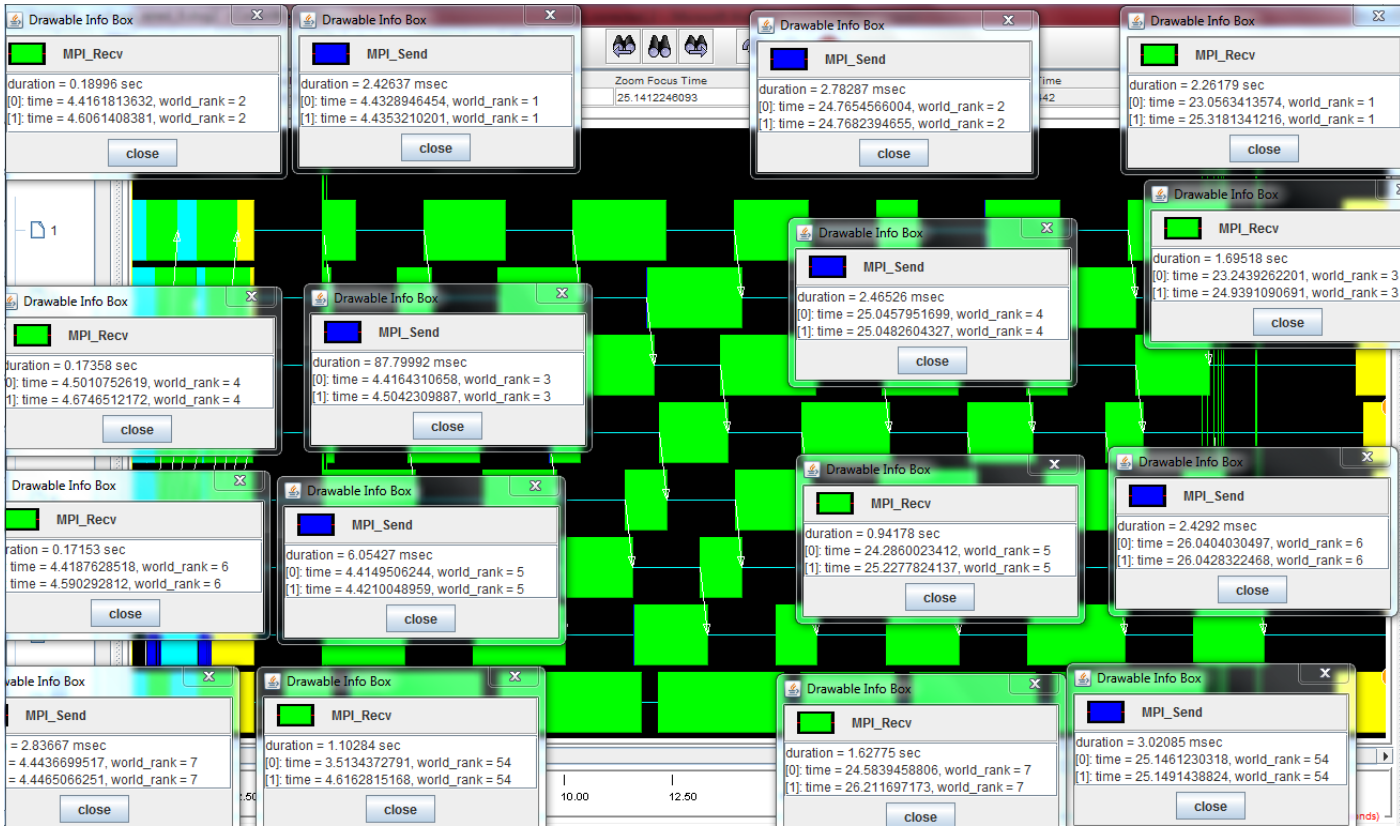


Figure 4.10:Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 8 cluster wired LAN

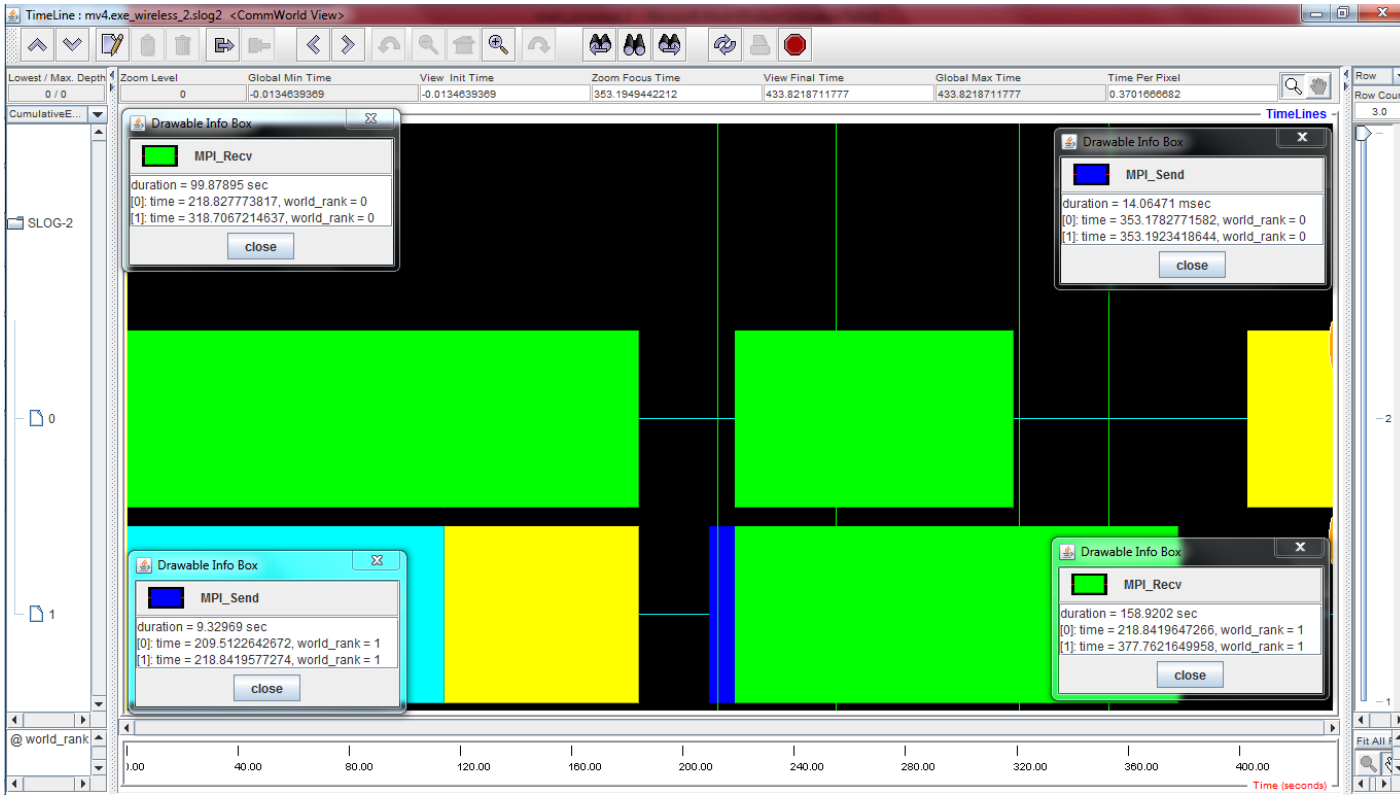


Figure 4.11: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 2 cluster wireless LAN

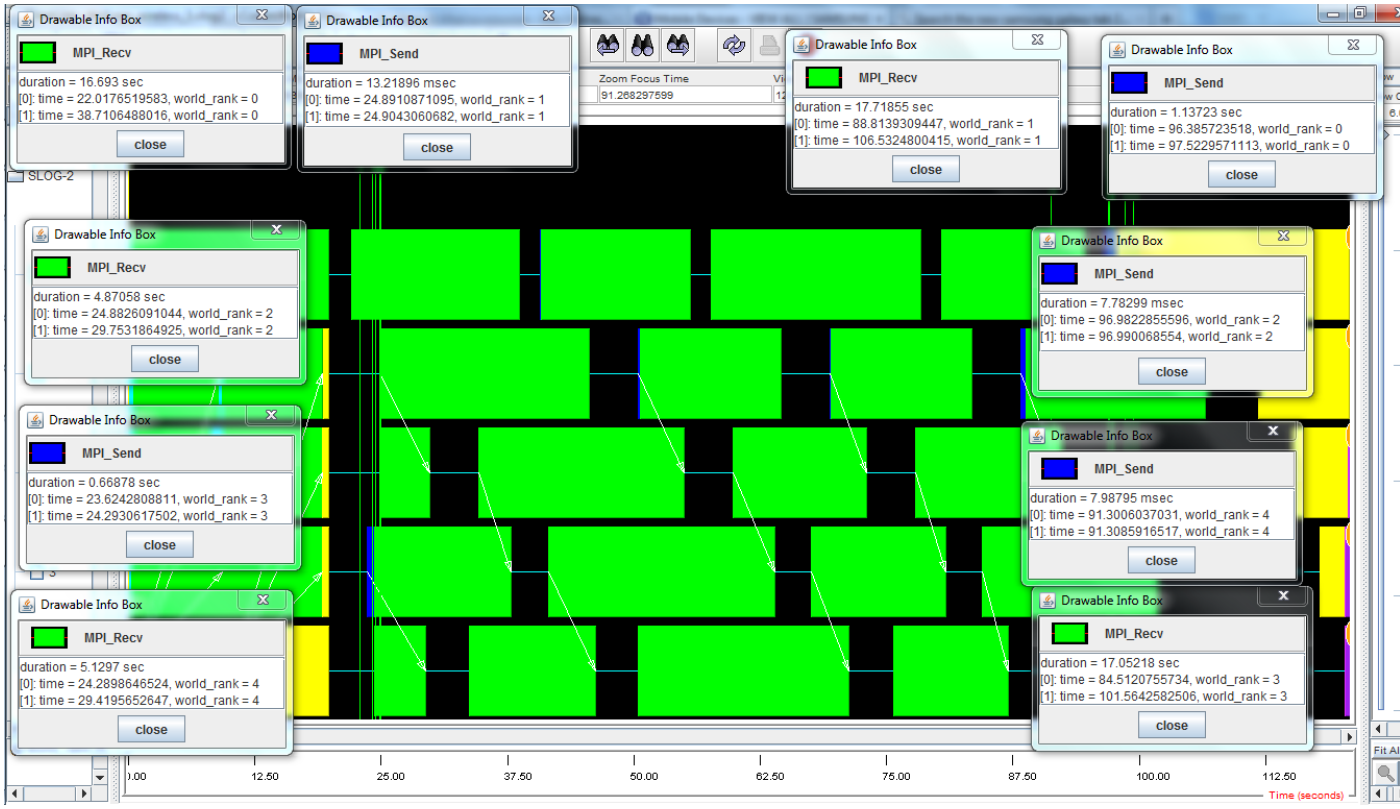


Figure 4.12: Jumpshot view of parallel matrix-matrix multiplication (row-wise decomposition) algorithm on 5 cluster wireless LAN

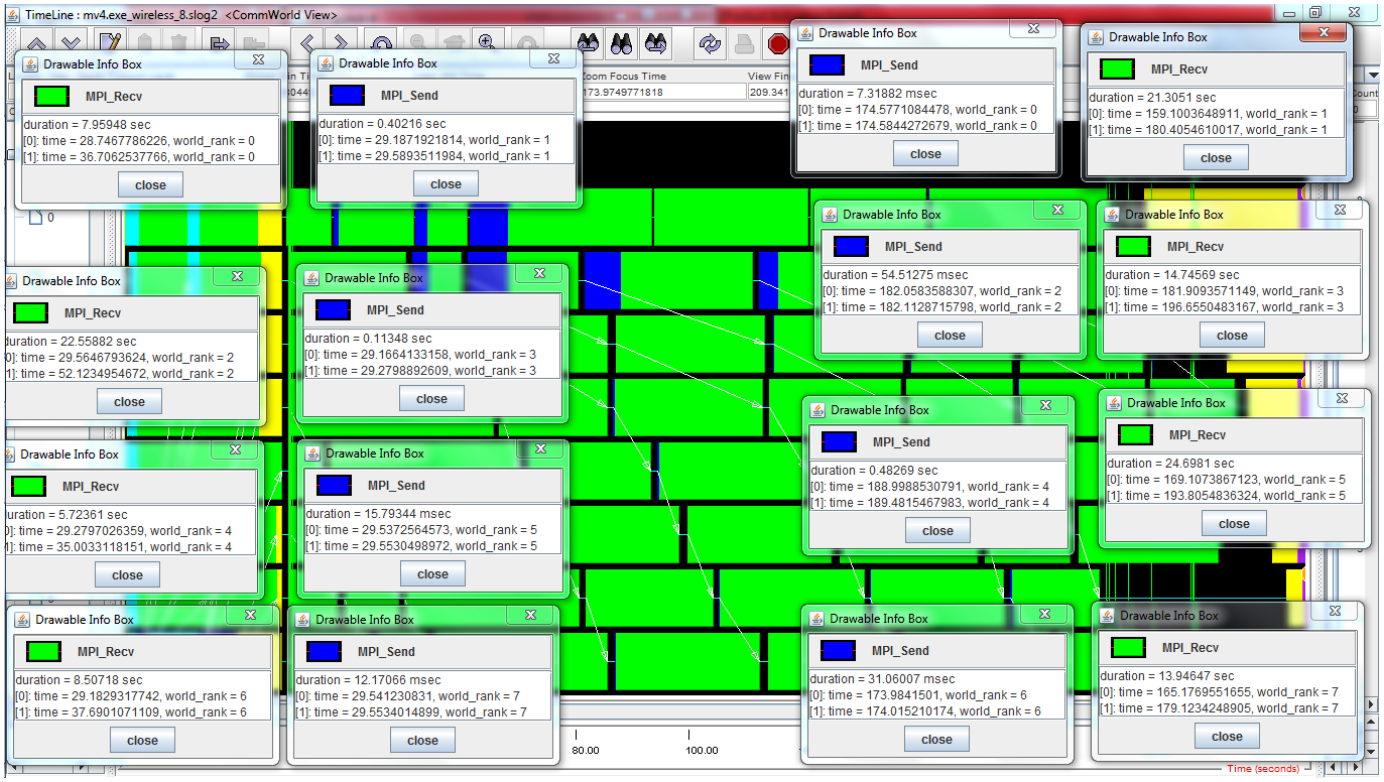


Figure 4.13: Jumpshot view of parallel Matrix-Matrix Multiplication (row-wise decomposition) algorithm on 8 cluster wireless LAN

4.8 Multiprogramming approach to performance improvement

Delays and media contention in wireless medium has been observed to be a major contributor to the degradation in performance of parallel programs on wireless network. Therefore, if there is a way to reduce collision in wireless medium, a significant improvement in performance is likely to be observed. This informed the multiprogramming approach. In this experiment, each system runs more than one process of the entire program,therefore localizing some of the communications among processes. The following subsections discuss these performance improvements and limits to the multiprogramming approach. The parallel matrix-matrix multiplication was used to show this improvement.This is because of its relatively higher computational complexity. This feature makes the parallel matrix-matrix multiplication algorithm more likely to show a clearer improvement in performance.

4.8.1 Multiprogramming:Two Processing Nodes

In Figure 4.14, performance was observed for twoto twenty twoprocesses, hence two to twenty two processes were shared equally among two systems. For each record taken, a total of six executions were made and average running time recorded.

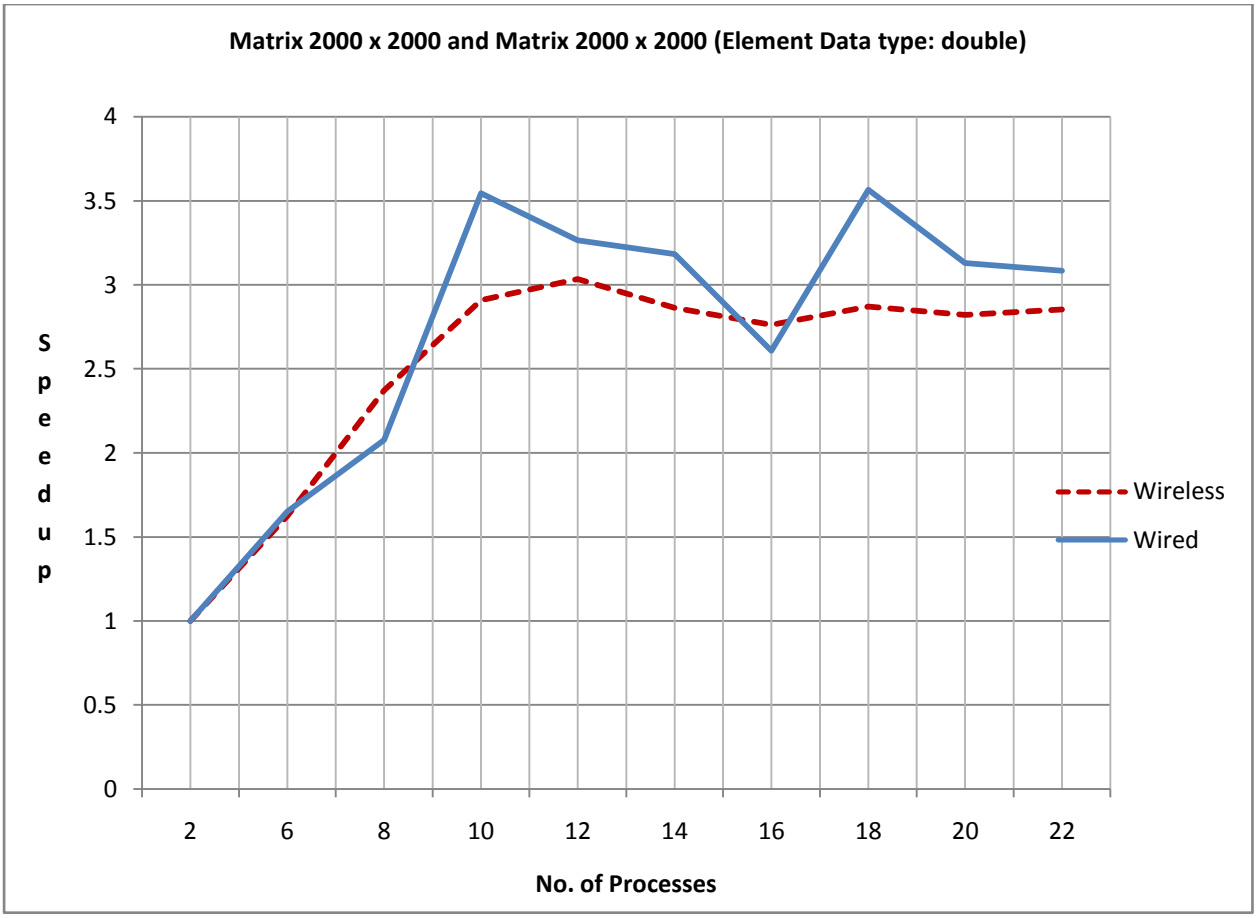


Figure 4.14: Performance graph for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 2

From the graph in Figure 4.14, it can be seen that performance increases up to a limit as the number of processes increases on each system. Overall performance is at its best since in effect only two network adapters are actually communicating over the wireless medium, all other processes are logically aligned within same host system.

4.8.2 Multiprogramming: Three Processing Nodes:

In Figure 4.15, performance was observed for three to thirty processes. Similar to the previous setup, here the number of processing nodes is three.Three to thirty processes were shared equality among three systems. For each record taken, a total of six executions were made and average running times recorded.

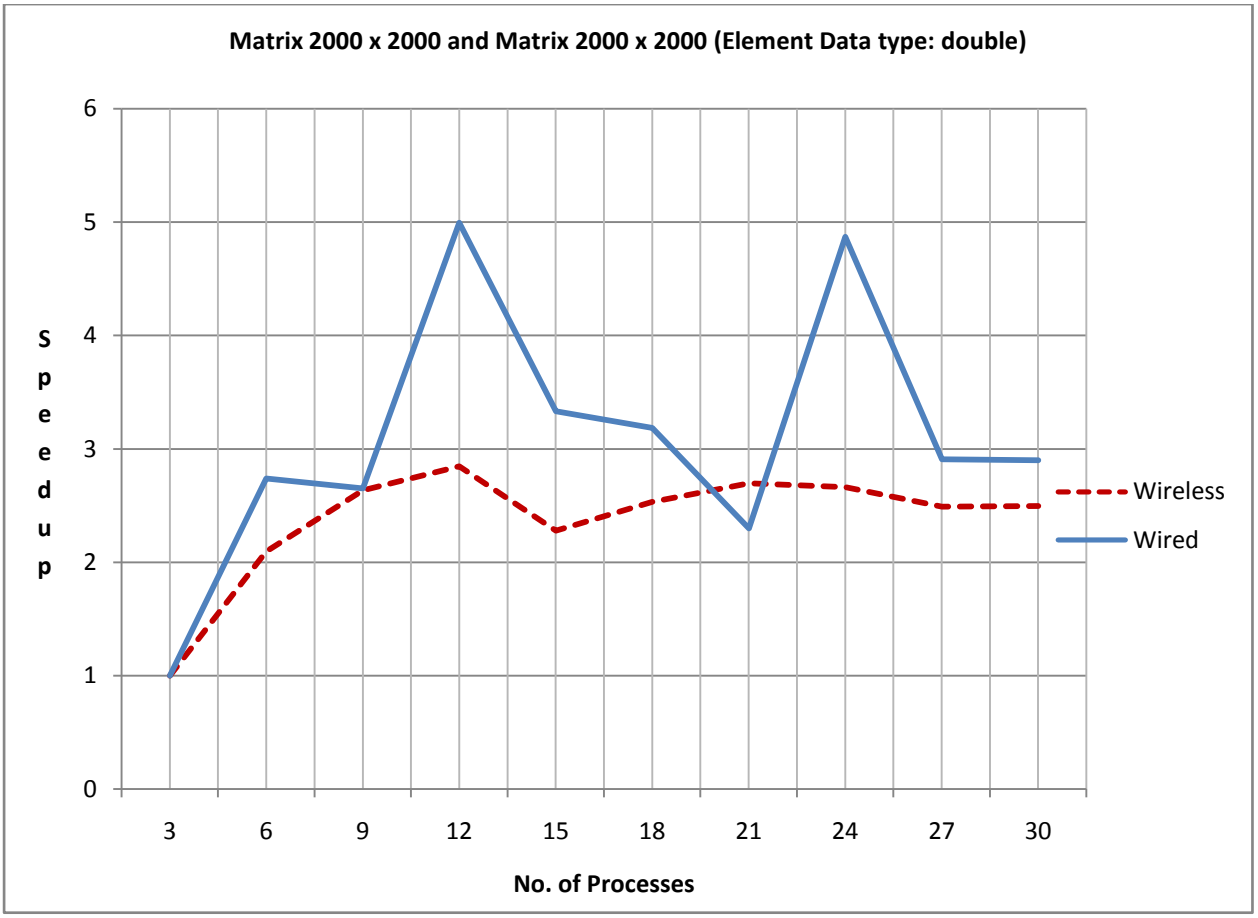


Figure 4.15: Performance for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 3

Figure 4.15 exposes very vital information on the performance of multiple processes. It should be observed that at special points which are multiples of the number of processing, the performance of the parallel program improves. Hence, the point where the number of processes is6, 12, 24, performance is observed to be better. This can be explained from the decomposition and the logarithmic reduction done by the parallel algorithm. It is observed that at points where the number of processing nodes divides the number of processes evenly, performance improves.

For example, at the point where the number of processes is 6, each of the three processing nodes has two processes. Therefore the number of processes per processing node is even so the performance peaks, also at the point where the number of processes is 12, this means that each processing node gets four processes. Again this is an even number of processes hence performance peaks at that point. However, at the point where the number of processes is 9, 15, 21 and 27, each processing node gets an odd number of processes and performance stalls. It is also observed that if the number of processing nodes is odd then at the point where the number of processes is an exponent factor of the processing node size, the performance is worse.

4.8.3 Multiprogramming: Four Processing Nodes

In Figure 4.16, performance was observed for four to forty processes. Similar to the previous setup, here the number of processing nodes is four. Therefore four to forty processes were shared equality among the four systems. For each record, a total of six executions were also made and average running timestaken.

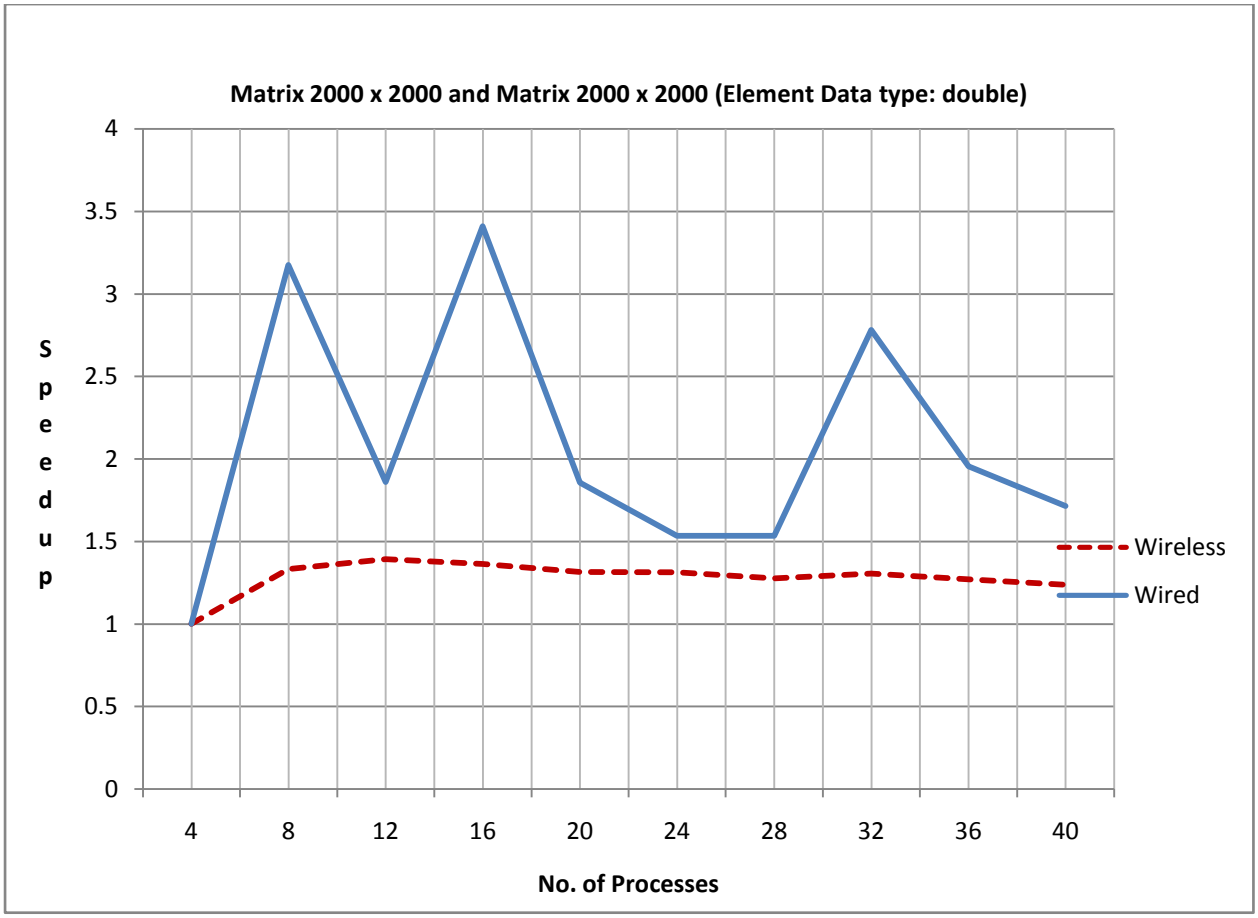


Figure 4.16: Performance for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 4

From Figure 4.16 it is clearly seen that the performance does not increase beyond 12 processes for four processing nodes. As observed in 4.7.2, the logarithmic effect of parallel reduction primitives which forms part of this algorithm makes performance to peak at certain point in the performance graph. These points are 8 and 16, and the explanation follows the one given in section 4.7.2.

4.8.4 Multiprogramming: FiveProcessing Nodes

In Figure 4.17, performance was observed for fiveto fifty processes.Therefore, five to fifty processes were shared among five systems. For each record taken, a total of six executions were made and average running times recorded.

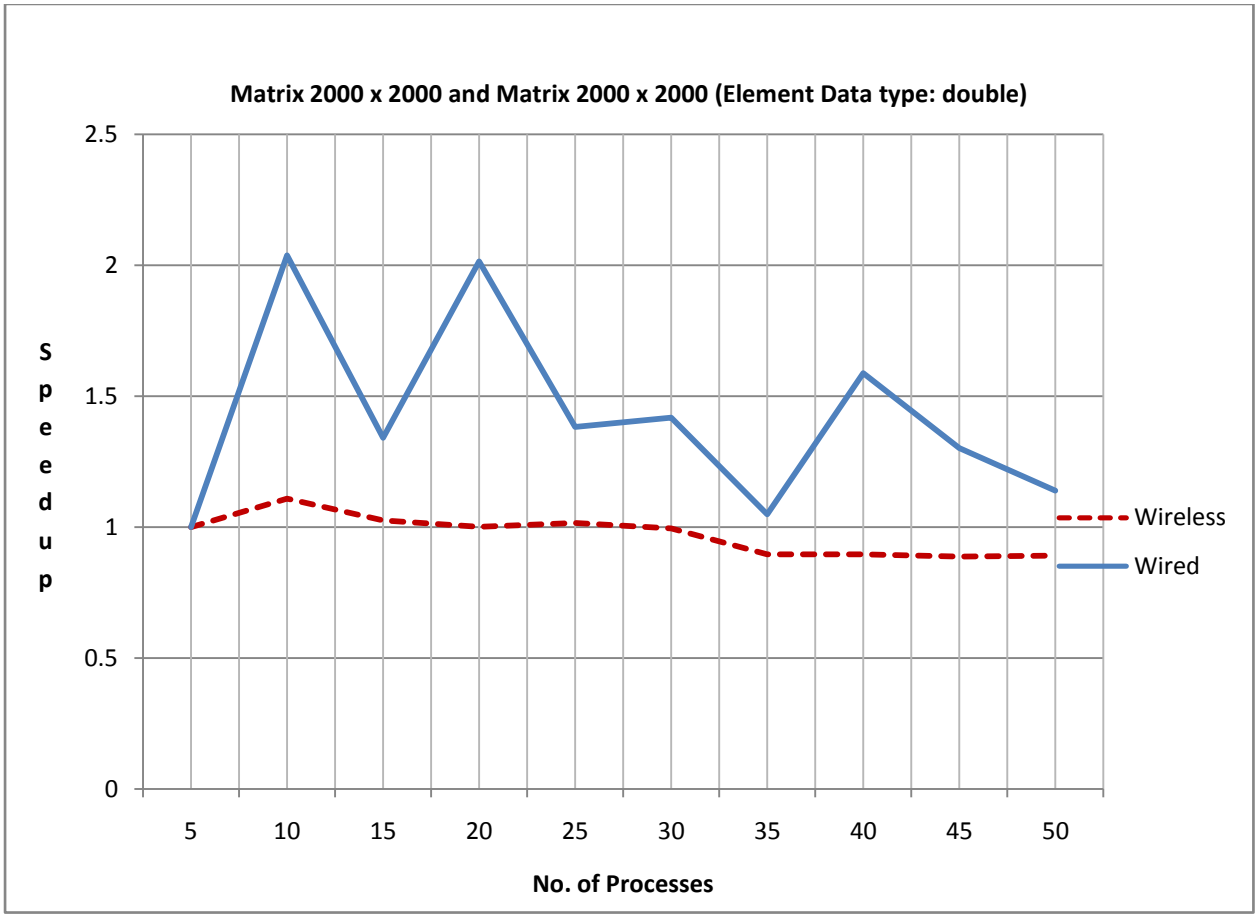


Figure 4.17: Performance graph for multiple processes - parallel matrix-matrix multiplication algorithm on cluster size 5

In Figure 4.17, the same performance similar to the previous two scenarios is seen but in this case the best performance is observed at the points where the number of processes is ten and shared among five processing nodes. The peak performance observed at points where the number of processes is 10, 20, 40 conforms with the explanation given for the previous experiment in section 4.6.3.

4.9 Chapter Summary

This chapter discussed the sequential matrix-matrix multiplication algorithm and the parallel matrix-matrix multiplication algorithm (row-wise decomposition). It explained the design analysis, algorithm time complexity and the results obtained from the matrix-matrix experiments. The chapter also showed how performance can be improved with the use of multiprogramming approach in parallel program execution on both wired and wireless LAN. It discusses the results of experiments on the same parallel matrix-matrix multiplication algorithm using the multiprogramming approach and discussed the limit to this approach.

CHAPTER FIVE

SUMMARY AND CONCLUSION

5.1 Summary

This thesis compares empirically,the performance of a typical communication intensive parallel program and a compute intensive parallel programs on two platforms, wired Ethernet network (Local Area Network,(LAN)) and wireless LAN(IEEE 802.11g) .

Four algorithms namely, Parallel Matrix-vector multiplication algorithm using row-wise decomposition, Parallel Matrix-vector multiplication algorithm using column-wise decomposition, Parallel Matrix-vector multiplication algorithm using checkerboard decomposition and Parallel Matrix-Matrix multiplication using row-wise decomposition were used as benchmark programs on Wired and Wireless LANs popularly known as Beowulf clusters.

The topology for both networks was Star topology. The Wired LAN was setup using Ethernet Cat 5 cables on Dlink switches while the wireless network was setup using a Dlink access point device.

In general it was observed that parallel programs on wireless network degrade as the number of processing node increases.

From a holistic point of view, this study has investigated whether the performance gain experienced by parallel programs in typical Beowulf clustersimplemented on wired network (LAN) can be replicated on a similar Beowulf cluster whose network of computers forms a wireless LAN. This question prompted the setup of the two

networks and the test carried out. Parallel Matrix-vector and parallel matrix-matrix multiplication with different data decomposition techniques like row-base, column-base and checkerboard were picked as the test algorithms because of the vast application of matrix multiplication algorithms in different computing programs/applications. These algorithms were also picked so as to give a clear distinctive result between algorithms that require high amount of computation and algorithms that require high amount of communication.

5.2 Conclusions

i. This experiment results showed that parallel programs written for Beowulf clusters on wireless LAN (IEEE 802.11) do not gain appreciable speedup as the number of processing nodes increases compared to the same parallel programs written for the same Beowulf clusters but on wired LAN.

   The ratio, in bandwidth speed, of a typical wired LAN and wireless LAN is 100:54 respectively and one of the research questions was to find out if parallel programs on these two networks exhibit these same differences in performance. It was observed that the speed ratio wasn't maintained for these algorithms. For example, the time it took the parallel matrix-matrix multiplication algorithm on wired cluster to complete its task was 97.01seconds while that of the wireless was 110.1seconds. As the processing node increased, the time it took the parallel program deployed on wireless network to complete its task increased while that of wired network decreased. This clearly showed that speed of the network isn't the only contributing factor to the degradation observed.

iii. The speedup/degradationin both wired and wireless networks respectively is due to the nature and protocols of both networks. While the wired network (fast Ethernet)running on full-duplex mode suffers no collision and as such requires no CSMA/CD protocol with random back-off time, the wireless LAN (IEEE 802.11g) is half-duplex with an Access point and each station shares the same medium of communication (i.e, electromagnetic wave). The blocking nature of the send and receive primitive couple with the random back-off time contribute to the increase in delay experienced by the send/receive primitives of the parallel programs hence nullifying the positive effect an increase in the number of processing node in a Beowulf cluster –deployed on a wireless LAN -would have on the overall speedup of parallel program.

iii. Since at the moment, collision cannot be totally eliminated in wireless LAN, the need to localize some of the processes led to experimenting with multiple processes per processing node. This led to a significant speedup experienced in both the wired and wireless LAN albeit with limitations on the number of processes that can be executed on each processing node. In general it was observed that performance is at its peak at points where the number of processes per processing node is even and if the Beowulf cluster size is odd then at the point where the number of processes is an exponential factor of the cluster size performance is worse.

5.3 Recommendation for Further Study

This thesis has discussed parallel program performance on local area networks with focus on fast Ethernet and IEEE 802.11g networks hence it will be worthwhile to consider the following:

i. Compare performance of parallel programs across the various sub-categories of the IEEE 802.11 networks namely IEEE 802.11a, IEEE802.11b, and IEEE 802.11n.

ii. Another area that will require an intense study will be in the randomness of the back-off timer of the CSMA/CA protocol. It will be worthwhile to test different random distribution functions to find one that is best suited for parallel programs on wireless LAN.

iii. And lastly, since collision is a major problem for Beowulf clusters in wireless networks, it will be worthwhile to experiment with a larger LAN consisting of several access points. This way the collision domain can be further broken down and performance may likely improve.

REFERENCES

Alfredo, B., Julien, L., Jakub, K., and Jack, D. (2012). *Parallel Tiled QR Factorization for Multicore LAPACK Working Note # 190.* Retrieved 10 07, 2012, from www.netlib.org: www.netlib.org/../lawn190.pdf

Ananth, G., Anshul, G., George, K. and Vipin, K. (January 16, 2003). *Introduction to Parallel Computing.* Addison Wesley, pp 1-14.

Arnab, S., and Nabanita, D. (2009). A Comparative Study of the MPI communication primitives on a cluster. *Journal of Communication and Computer*, vol 15, pp.15-19.

Behrooz, P. (2002). *Introduction to parallel processing algorithms and architectures.* New York: Kluwer Academic Publishers, pp 3-47.

Behrouz, A. F., and Sophia , C. F. (2007). *Data Communication and Networking.* New York: McGraw Hill, pp 395, pp 421.

Brown, N. E. (2010). *Type Oriented Parallel Programming. Doctoral Thesis, Durham University.* Retrieved Retrieved 09 24, 2012, from http://etheses.dur.ac.uk/106

Cameron, H. and Tracey, H. (2003). *Parallel and Distributed Programming in C++*. Addison Wesley.

Christoph, K. and Jorg, K. (2007). *Models for Parallel Computing, Review and Perspectives.* Retrieved 09 24, 2012, from http://www.ida.liu.se: http://www.ida.liu.se/~chrke/papers/modelsurvey.pdf

DeinoMPI. (2009). DeinoMPI DocumenationRetrieved 09 24, 2012, from http://www.mpi.deino.net

DeinoMPI (2013). *deinompi*. Retrieved 01 20, 2013, from http://www.mpi.deino.net

Dharma, P. A. and Zeng, Q.-a. (2011). *Introduction to Wireless & Mobile Systems.* Stamford: Cengage Learning, pp 432-480

Flynn, M. (1972). *Some Computer Organizations and Their Effectiveness*. IEEE Trans. Comput. C–21: 948

Gropp, W., Ewing , L. and Thomas, S. (2003). *Beowulf Cluster Computing with Linux.* Massachusetts London: The MIT Press.

Intel. (2009). Intel Cilk++ SDK Programmer's Guide.Retrieved 09 24, 2012, from www.intel.com

Intel Thread Building Block. (2013). *Intel Threading Building Blocks -Intel TBB 4.1*. Retrieved 01 28, 2013, from http://software.intel.com/en-us/intel-tbb

John, L. H. and David, A. P. (2007). *Computer Architecture, A Quantitative Approach 4th Edition.* Amsterdam: Morgan Kaufmann Publisher.

Linda, N. and Lobur, J. (2006). *The Essentials of Computer Organization and Archictecture.* London: Jones and Bartlett Publishers, Inc, pp 15-78

Michael J., Q. (2003). *Parallel Programming in C with MPI and OpenMP.* New York: McGraw-Hill, pp 3-215

MPICH. (2013). *MPICH*. Retrieved 01 23, 2013, from http://www.mpich.org

Message Passing Interface (MPI). (2013). *Standard Message Passing interface MPI.* Retrieved 01 28, 2013, from http://www.mcs.anl.gov/research/projects/mpi/

Pawel, G. and Michal, F. K. (2006). Multi-Core Processors: New Way to Achieve High System Performance. *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06).*

Rahul, I. M., Vikas, G. and R., K. Bansal. (2011). Simulation & Performance Analysis of Wired and Wireless. *Global Journal Of Computer Science And Technology*, 14(7):11- 17

Ravela, S. C. (2010). Comparison of Shared Memory based parallel programming models: Thesis Report. Blekinge Institute of Technology, Ronneby, Sweden.

Rinat, K., Ahmed, P. and H.D.J. (2012). *Distributed Parallel Computing in Networks of Workstations : A survey Study.* Retrieved September 24, 2012, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.9.3661&rep=rep1&type=pdf

Samarth, H. S., Kai, C. and Klara, N. (2013). *Available Bandwidth Estimation in IEEE 802.11-based.* Retrieved 04 18, 2013, from www.caida.org/outreach/isma/0312/abstracts/shah.pdf

Stephen, L. O. and Jan, F. P. (2010). Comparison of OpenMP 3.0 and Other Task Parallel, Frameworks on Unbalanced Task Graphs. *International Journal of Parallel Computing*, 38:341–360.

Thomus, H. C., Charles, E. L., Ronald, L. R. and Clifford, S. (2002). *Introduction to Algorithms.* Massachussets: MIT Press, pp 41-91.

APPENDIX A

Source Code for Parallel Matrix-Matrix Algorithm, contributed by author of this thesis

```
/*
 *  Matrix-Matrix multiplication, Version 1
 *
 *  This program multiplies a matrix and a matrix input from
 *  separate files. The result matrix is printed to standard
 *  output.
 *
 *  Data distribution of matrix: rowwise block striped on the two matrices
 *  Data distribution of matrix: every task exchange rows of matrix b in a cyclic form
 *  Data Distribution of matrix: resultant matrix is gathered at process p-1
 *
 *  Written/Programmed by Charles I. Saidu
 *        with  MyMpi.h and MyMPI.c written by Michael J. Quinn
 *        row-based matrix-matrix multiplication
 *  Last modification: 17 November 2012
 */

#include <stdio.h>
#include "mpi.h"
#include "MyMpi.h"
#include <stdlib.h> // Charles I. Saidu

typedef int dtype;
#define mpitype MPI_INT

void internalMult2(dtype *C, int locationI, int locationJ, int lowIndexB, dtype **a, dtype **b, int rowsA, int
        rowsB);
void internalMult1(dtype *C, int locationI, int locationJ, int lowIndexB, dtype *A, dtype *B, int cols, int
        rowsB);
void printArray(dtype *array, int row, int col);
void copyArrays(dtype *from, dtype *to, int length);
int main(int argc, char* argv[]){
        dtype **a; //points of pointers to the first address of rows in matrixA
        dtype **b; //points of pointers to the first address of rows in matrixA
        dtype **c; //points of pointers to the first address of rows in matrixA

        dtype *matrixA; //matrix A
        dtype *matrixB; //matrix B
        dtype *matrixC; //resultant matrix after matrix a and matrix multiplication
        dtype *partialC;

        dtype *sendBuffer;
        dtype *receiveBuffer;
        int *receive_cnt;
        int *receive_disp;

        int mA, nA, mB, nB; //columns and rows for matrices for matrices A and B
        int  id, p; //process id and number of processes p
        int maxBlock; //maximum allocated block
        int rowsA;  //number of allocation rows for Matrix A
        int rowsB;          //number of allocation rows for Matrix B
        double seconds; //timer for benchmarking speeding
        double max_seconds; //maximum seconds elapsed for all the jobs
        int    processes_per_host;
```

```
	char name[BUFSIZ];
	int length;


	FILE *outFile;
	outFile = fopen("mv4Out.out", "a");
	if (!outFile){
	printf("Cannot open output file.\n");
	exit(2);
	}
	processes_per_host=atoi(argv[3]);


	MPI_Init (&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	MPI_Get_processor_name(name, &length);

	//Read and distribute matrix A
	read_row_striped_matrix (argv[1], (void *) &a,
		(void *) &matrixA, mpitype, &mA, &nA, MPI_COMM_WORLD);


	//Read and distribute matrix B
	read_row_striped_matrix (argv[2], (void *) &b,
		(void *) &matrixB, mpitype, &mB, &nB, MPI_COMM_WORLD);


	//number of stripped rows from matrix A and Matrix B
	rowsB = BLOCK_SIZE(id,p,mB);
	rowsA=BLOCK_SIZE(id, p, mA);

	//index boundaries for stripped rows and
	int lowA = BLOCK_LOW(id, p, mA);
	int highA = BLOCK_HIGH(id, p, mA);
	int lowB = BLOCK_LOW(id, p, mB);
	int highB = BLOCK_HIGH(id, p, mB);


//resultant matrix intialization
	if(id==(p-1)){
		matrixC=(dtype *)malloc(mA*nB*sizeof(dtype));
	}

	//partial matrix initialization
	partialC=(dtype *)malloc(rowsA*nB*sizeof(dtype));

	int j, i, k, l, y;
	//Multiple initialize partial matrix to all zeros
	for(j=0; j<(rowsA*nB); j++){
		partialC[j]=0;
	}

	maxBlock = BLOCK_SIZE((p-1), p, mB); //because the maximum block is assigned to process p-1
	according to decomposition function
	int bufferSize =((maxBlock*nB)+2); //buffer size for stripped matrix about to be exchanged among
	processes in a cyclic form

	sendBuffer=(dtype *) malloc( bufferSize * sizeof(dtype) ); //maximum block sendBuffer memory
	allocation
```

```
receiveBuffer=(dtype *) malloc( bufferSize * sizeof(dtype) ); //maximum block receiveBuffer memory allocation


//set value of low and number of rows of submatrix B before sending sub matrix to process i+1
sendBuffer[0]=lowB;
sendBuffer[1]=rowsB;

//copy the content of submatrix B to sendBuffer and receive buffer for cyclic exchange of data between processes
for(i=2; i<(bufferSize); i++){
	sendBuffer[i]=matrixB[i-2];
	receiveBuffer[i]=matrixB[i-2];
}

//benchmark running time begins here
MPI_Barrier (MPI_COMM_WORLD);
seconds = - MPI_Wtime();



//multiple process i's own share of stripped matrix block B
for(i=0; i< rowsA; i++){
	for(j=0; j<nA; j++){
			internalMult2(&partialC[0], i, j, lowB, (dtype **)a, (dtype **)b, nA, rowsB);
		}
}

//assign the low index and high index for submatrix b
//copy content of matrix b into buffer

//begin sending and receiving submatrix from p-2 processes
//maximum block for buffer

int sendCount=0;
while(sendCount<(p-1)){
	//Receive data from a lower ranked process.also takes care of deadlock when process rank is 0
	if((id%2)==0){
		//printf("\nprocess %d is even numbered", id);
		if(id==0){
			MPI_Recv(&receiveBuffer[0], bufferSize, mpitype, (p-1), 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		else{
			MPI_Recv(&receiveBuffer[0], bufferSize, mpitype, (id-1), 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
		}
		//perform multiplication of recieved blocks
		for(i=0; i< rowsA; i++){
				for(j=0; j<nA; j++){
					internalMult1(&partialC[0], i, j, receiveBuffer[0], &matrixA[0], &receiveBuffer[2], nA, receiveBuffer[1]);
				}
		}
		MPI_Send(&sendBuffer[0], bufferSize, mpitype, (id+1)%p, 0, MPI_COMM_WORLD);
		copyArrays(&receiveBuffer[0], &sendBuffer[0], bufferSize);
	}
	else{
		//printf("\nprocess %d is odd numbered", id);
```

```
                MPI_Send(&sendBuffer[0], bufferSize, mpitype, (id+1)%p, 0,
        MPI_COMM_WORLD);
                MPI_Recv(&receiveBuffer[0], bufferSize, mpitype, (id-1), 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                for(i=0; i< rowsA; i++){
                        for(j=0; j<nA; j++){
                                internalMult1(&partialC[0], i, j, receiveBuffer[0],
        &matrixA[0], &receiveBuffer[2], nA, receiveBuffer[1]);
                        }
                }

                copyArrays(&receiveBuffer[0], &sendBuffer[0], bufferSize);
            }
            sendCount++;
        }

        //prepare receive_cnt vector and displacement vector for MPI_AllGatherV
        receive_cnt=(int *)malloc(p*sizeof(int));
        receive_disp=(int *)malloc(p*sizeof(int));

        for(i=0; i<p; i++){
            int blockP=BLOCK_SIZE(i,p,mB); //receive count for process i
            receive_cnt[i]=blockP*mB;  //assign receive count for process i
            int disp=BLOCK_LOW(i,p,mB); //calculate displacement for process i
            receive_disp[i]=disp*mB; //assign displacement for process i
        }

        //performance MPI gather operation and to process (p-1)
        MPI_Gatherv(&partialC[0], rowsA*nB, mpitype, &matrixC[0],&receive_cnt[0], &receive_disp[0],
        mpitype, (p-1), MPI_COMM_WORLD);

        MPI_Barrier (MPI_COMM_WORLD);
        seconds += MPI_Wtime();
        MPI_Reduce (&seconds, &max_seconds, 1, MPI_DOUBLE, MPI_MAX, 0,MPI_COMM_WORLD);
        //MPI_Allreduce (&seconds, &max_seconds, 1, MPI_DOUBLE, MPI_MAX,
        MPI_COMM_WORLD);
        if(!id)
        {
            fprintf (outFile, "%d\t%d\t%12.6f\t%d\n", processes_per_host, p, max_seconds, mA);

        }

        if(id==(p-1)){
        //      printf("\n Resultant Matrix \n");
        //      printArray(&matrixC[0], mA, nB);
            free(matrixC); //matrixC is only initialized in process p-1
        }
        free(partialC);
        free(sendBuffer);
        free(receiveBuffer);
        fflush(stdin);
fflush(stdout);
        MPI_Finalize();
        return 0;
 }
void internalMult2(dtype *C, int locationI, int locationJ, int lowIndexB, dtype **a, dtype **b, int cols, int
        rowsB){
            int i, j,k;
            dtype sum=0;
            k=locationI;
```

```
		for(i=0; i<rowsB; i++){
			sum+=a[locationI][lowIndexB]*b[i][locationJ];
			lowIndexB++;
		}
		C[((locationI*cols)+locationJ)]+=sum;
	}
void internalMult1(dtype *C, int locationI, int locationJ, int lowIndexB, dtype *A, dtype *B, int cols, int
	rowsB){
		int i, j,k;
		dtype sum=0;
		k=locationI;
		for(i=0; i<rowsB; i++){
			sum+=A[((locationI*cols)+lowIndexB)]*B[((i*cols)+locationJ)];
			lowIndexB++;
		}
		C[((locationI*cols)+locationJ)]+=sum;
	}

void copyArrays(dtype *from, dtype *to, int length){
int i;
		for(i=0 ; i<length; i++){
			to[i]=from[i];
		}
	}

void printArray(dtype *array, int row, int col){
	int i, j;
	printf("\n");
	for(i=0; i<row; i++){
		for(j=0; j<col; j++){
			printf("%d\t", array[(i*col)+j]);
		}
		printf("\n");
	}
 }
```

APPENDIX B

MPI Headers and Library functions

MyMPI.h and MyMPI.c libraries written by Michael J. Quinn

```
/*   MyMPI.h
 *
 *   Header file for a library of matrix/vector
 *   input/output/redistribution functions.
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 4 September 2002
 */
/************************** MACROS ***************************/
#define DATA_MSG          0
#define PROMPT_MSG        1
#define RESPONSE_MSG      2

#define OPEN_FILE_ERROR    -1
#define MALLOC_ERROR       -2
#define TYPE_ERROR         -3

#define MIN(a,b)          ((a)<(b)?(a):(b))
#define BLOCK_LOW(id,p,n)  ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n) \
                (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(j,p,n) (((p)*((j)+1)-1)/(n))
#define PTR_SIZE          (sizeof(void*))
#define CEILING(i,j)      (((i)+(j)-1)/(j))

/****************** MISCELLANEOUS FUNCTIONS ******************/

void  terminate (int, char *);

/*************** DATA DISTRIBUTION FUNCTIONS ***************/

void replicate_block_vector (void *, int, void *,
      MPI_Datatype, MPI_Comm);
void create_mixed_xfer_arrays (int, int, int, int**, int**);
void create_uniform_xfer_arrays (int, int, int, int**,int**);

/******************* INPUT FUNCTIONS ************************/

void read_checkerboard_matrix (char *, void ***, void **,
      MPI_Datatype, int *, int *, MPI_Comm);
void read_col_striped_matrix (char *, void ***, void **,
      MPI_Datatype, int *, int *, MPI_Comm);
void read_row_striped_matrix (char *, void ***, void **,
```

```
      MPI_Datatype, int *, int *, MPI_Comm);
void read_block_vector (char *, void **, MPI_Datatype,
int *, MPI_Comm);
void read_replicated_vector (char *, void **, MPI_Datatype,
int *, MPI_Comm);

/****************** OUTPUT FUNCTIONS ***********************/

void print_checkerboard_matrix (void **, MPI_Datatype, int,
int, MPI_Comm);
void print_col_striped_matrix (void **, MPI_Datatype, int,
int, MPI_Comm);
void print_row_striped_matrix (void **, MPI_Datatype, int,
int, MPI_Comm);
void print_block_vector (void *, MPI_Datatype, int,
      MPI_Comm);
void print_replicated_vector (void *, MPI_Datatype, int,
      MPI_Comm);



/*
 *   MyMPI.c -- A library of matrix/vector
 *   input/output/redistribution functions
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 4 September 2002

 *           Modified by Charles I. Saidu
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include "MyMPI.h"
/****************** MISCELLANEOUS FUNCTIONS *****************/
/*
 *   Given MPI_Datatype 't', function 'get_size' returns the
 *   size of a single datum of that data type.
```

```
 */
int get_size (MPI_Datatype t) {
if (t == MPI_BYTE) return sizeof(char);
if (t == MPI_DOUBLE) return sizeof(double);
if (t == MPI_FLOAT) return sizeof(float);
if (t == MPI_INT) return sizeof(int);
if (t == MPI_SHORT) return sizeof(short);

printf ("Error: Unrecognized argument to 'get_size'\n");
fflush (stdout);
  MPI_Abort (MPI_COMM_WORLD, TYPE_ERROR);
}
/*
 *   Function 'my_malloc' is called when a process wants
 *   to allocate some space from the heap. If the memory
 *   allocation fails, the process prints an error message
 *   and then aborts execution of the program.
 */
void *my_malloc (
int id,     /* IN - Process rank */
int bytes)  /* IN - Bytes to allocate */
{
void *buffer;
if ((buffer = malloc ((size_t) bytes)) == NULL) {
printf ("Error: Malloc failed for process %d\n", id);
fflush (stdout);
     MPI_Abort (MPI_COMM_WORLD, MALLOC_ERROR);
  }
return buffer;
}
/*
 *   Function 'terminate' is called when the program should
 *   not continue execution, due to an error condition that
```

```
 *   all of the processes are aware of. Process 0 prints the
 *   error message passed as an argument to the function.
 *
 *   All processes must invoke this function together!
 */
void terminate (
int   id,           /* IN - Process rank */
char *error_message) /* IN - Message to print */
{
if (!id) {
printf ("Error: %s\n", error_message);
fflush (stdout);
   }
  MPI_Finalize();
exit (-1);
}
/************ DATA DISTRIBUTION FUNCTIONS ******************/
/*
 *   This function creates the count and displacement arrays
 *   needed by scatter and gather functions, when the number
 *   of elements send/received to/from other processes
 *   varies.
 */
void create_mixed_xfer_arrays (
int id,       /* IN - Process rank */
int p,        /* IN - Number of processes */
int n,        /* IN - Total number of elements */
int **count,  /* OUT - Array of counts */
int **disp)   /* OUT - Array of displacements */
{
int i;
   *count = my_malloc (id, p * sizeof(int));
   *disp = my_malloc (id, p * sizeof(int));
```

```
   (*count)[0] = BLOCK_SIZE(0,p,n);
   (*disp)[0] = 0;
for (i = 1; i < p; i++) {
      (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
      (*count)[i] = BLOCK_SIZE(i,p,n);
   }
}
/*
 *   This function creates the count and displacement arrays
 *   needed in an all-to-all exchange, when a process gets
 *   the same number of elements from every other process.
*/
void create_uniform_xfer_arrays (
int id,        /* IN - Process rank */
int p,         /* IN - Number of processes */
int n,         /* IN - Number of elements */
int **count,   /* OUT - Array of counts */
int **disp)    /* OUT - Array of displacements */
{
int i;
   *count = my_malloc (id, p * sizeof(int));
   *disp = my_malloc (id, p * sizeof(int));
   (*count)[0] = BLOCK_SIZE(id,p,n);
   (*disp)[0] = 0;
for (i = 1; i < p; i++) {
      (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
      (*count)[i] = BLOCK_SIZE(id,p,n);
   }
}
/*
 *   This function is used to transform a vector from a
 *   block distribution to a replicated distribution within a
 *   communicator.
```

```
 */
void replicate_block_vector (
void        *ablock,  /* IN - Block-distributed vector */
int         n,       /* IN - Elements in vector */
void        *arep,    /* OUT - Replicated vector */
  MPI_Datatype dtype,   /* IN - Element type */
  MPI_Comm     comm)    /* IN - Communicator */
{
int *cnt;  /* Elements contributed by each process */
int *disp; /* Displacement in concatenated array */
int id;    /* Process id */
int p;     /* Processes in communicator */
  MPI_Comm_size (comm, &p);
  MPI_Comm_rank (comm, &id);
  create_mixed_xfer_arrays (id, p, n, &cnt, &disp);
  MPI_Gatherv (ablock, cnt[id], dtype, arep, cnt, disp, dtype, 0,comm);
  //MPI_Allgatherv (ablock, cnt[id], dtype, arep, cnt, disp, dtype, comm);
free (cnt);
free (disp);
}
/********************** INPUT FUNCTIONS **********************/
/*
 *   Function 'read_checkerboard_matrix' reads a matrix from
 *   a file. The first two elements of the file are integers
 *   whose values are the dimensions of the matrix ('m' rows
 *   and 'n' columns). What follows are 'm'*'n' values
 *   representing the matrix elements stored in row-major
 *   order.  This function allocates blocks of the matrix to
 *   the MPI processes.
 *
 *   The number of processes must be a square number.
 */
void read_checkerboard_matrix (
```

```
char *s,              /* IN - File name */
void ***subs,         /* OUT - 2D array */
void **storage,       /* OUT - Array elements */
  MPI_Datatype dtype,   /* IN - Element type */
int *m,               /* OUT - Array rows */
int *n,               /* OUT - Array cols */
  MPI_Comm grid_comm)   /* IN - Communicator */
{
void     *buffer;         /* File buffer */
int      coords[2];      /* Coords of proc receiving
next row of matrix */
int      datum_size;     /* Bytes per elements */
int      dest_id;        /* Rank of receiving proc */
int      grid_coord[2];  /* Process coords */
int      grid_id;        /* Process rank */
int      grid_period[2]; /* Wraparound */
int      grid_size[2];   /* Dimensions of grid */
int      i, j, k;
  FILE     *infileptr;     /* Input file pointer */
void     *laddr;          /* Used when proc 0 gets row */
int      local_cols;     /* Matrix cols on this proc */
int      local_rows;     /* Matrix rows on this proc */
void     **lptr;          /* Pointer into 'subs' */
int      p;              /* Number of processes */
void     *raddr;          /* Address of first element
to send */
void     *rptr;          /* Pointer into 'storage' */
  MPI_Status status;         /* Results of read */

  MPI_Comm_rank (grid_comm, &grid_id);
  MPI_Comm_size (grid_comm, &p);
  datum_size = get_size (dtype);
  /* Process 0 opens file, gets number of rows and
```

```
number of cols, and broadcasts this information
to the other processes. */
if (grid_id == 0) {
infileptr = fopen (s, "r");
if (infileptr == NULL) *m = 0;
else {
fread (m, sizeof(int), 1, infileptr);
fread (n, sizeof(int), 1, infileptr);
      }
   }
   MPI_Bcast (m, 1, MPI_INT, 0, grid_comm);

if (!(*m)) MPI_Abort (MPI_COMM_WORLD, OPEN_FILE_ERROR);

   MPI_Bcast (n, 1, MPI_INT, 0, grid_comm);
   /* Each process determines the size of the submatrix
it is responsible for. */
   MPI_Cart_get (grid_comm, 2, grid_size, grid_period,
      grid_coord);
   local_rows = BLOCK_SIZE(grid_coord[0],grid_size[0],*m);
   local_cols = BLOCK_SIZE(grid_coord[1],grid_size[1],*n);

   /* Dynamically allocate two-dimensional matrix 'subs' */
   *storage = my_malloc (grid_id,
      local_rows * local_cols * datum_size);
   *subs = (void **) my_malloc (grid_id,local_rows*PTR_SIZE);
lptr = (void *) *subs;
rptr = (void *) *storage;
for (i = 0; i < local_rows; i++) {
      *(lptr++) = (void *) rptr;
rptr += local_cols * datum_size;
   }
   /* Grid process 0 reads in the matrix one row at a time
```

```
and distributes each row among the MPI processes. */
if (grid_id == 0)
buffer = my_malloc (grid_id, *n * datum_size);
   /* For each row of processes in the process grid... */
for (i = 0; i < grid_size[0]; i++) {
coords[0] = i;
     /* For each matrix row controlled by this proc row...*/
for (j = 0; j < BLOCK_SIZE(i,grid_size[0],*m); j++) {
       /* Read in a row of the matrix */
if (grid_id == 0) {
fread (buffer, datum_size, *n, infileptr);
       }
       /* Distribute it among process in the grid row */
for (k = 0; k < grid_size[1]; k++) {
coords[1] = k;
          /* Find address of first element to send */
raddr = buffer +
            BLOCK_LOW(k,grid_size[1],*n) * datum_size;
          /* Determine the grid ID of the process getting
the subrow */
          MPI_Cart_rank (grid_comm, coords, &dest_id);
          /* Process 0 is responsible for sending...*/
if (grid_id == 0) {
            /* It is sending (copying) to itself */
if (dest_id == 0) {
laddr = (*subs)[j];
memcpy (laddr, raddr,
                 local_cols * datum_size);
            /* It is sending to another process */
            } else {
              MPI_Send (raddr,
                BLOCK_SIZE(k,grid_size[1],*n), dtype,
              dest_id, 0, grid_comm);
```

```
            }
          /* Process 'dest_id' is responsible for
receiving... */
          } else if (grid_id == dest_id) {
            MPI_Recv ((*subs)[j], local_cols, dtype, 0,
              0, grid_comm,&status);
          }
        }
      }
  }
if (grid_id == 0) free (buffer);
}
/*
 *   Function 'read_col_striped_matrix' reads a matrix from a
 *   file.  The first two elements of the file are integers
 *   whose values are the dimensions of the matrix ('m' rows
 *   and 'n' columns).  What follows are 'm'*'n' values
 *   representing the matrix elements stored in row-major
 *   order.  This function allocates blocks of columns of the
 *   matrix to the MPI processes.
 */
void read_col_striped_matrix (
char         *s,       /* IN - File name */
void      ***subs,     /* OUT - 2-D array */
void       **storage,  /* OUT - Array elements */
     MPI_Datatype dtype,    /* IN - Element type */
int         *m,        /* OUT - Rows */
int         *n,        /* OUT - Cols */
     MPI_Comm     comm)     /* IN - Communicator */
{
void      *buffer;        /* File buffer */
int        datum_size;    /* Size of matrix element */
int        i, j;
```

```
int        id;           /* Process rank */
  FILE      *infileptr;     /* Input file ptr */
int        local_cols;    /* Cols on this process */
void      **lptr;          /* Pointer into 'subs' */
void       *rptr;          /* Pointer into 'storage' */
int        p;             /* Number of processes */
int       *send_count;    /* Each proc's count */
int       *send_disp;     /* Each proc's displacement */
  MPI_Comm_size (comm, &p);
  MPI_Comm_rank (comm, &id);
  datum_size = get_size (dtype);
  /* Process p-1 opens file, gets number of rows and
cols, and broadcasts this info to other procs. */

if (id == (p-1)) {
infileptr = fopen (s, "r");
if (infileptr == NULL) *m = 0;
else {
fread (m, sizeof(int), 1, infileptr);
fread (n, sizeof(int), 1, infileptr);
     }
  }
  MPI_Bcast (m, 1, MPI_INT, p-1, comm);
if (!(*m)) MPI_Abort (comm, OPEN_FILE_ERROR);
  MPI_Bcast (n, 1, MPI_INT, p-1, comm);
  local_cols = BLOCK_SIZE(id,p,*n);
  /* Dynamically allocate two-dimensional matrix 'subs' */
  *storage = my_malloc (id, *m * local_cols * datum_size);
  *subs = (void **) my_malloc (id, *m * PTR_SIZE);
lptr = (void *) *subs;
rptr = (void *) *storage;
for (i = 0; i < *m; i++) {
     *(lptr++) = (void *) rptr;
```

```
rptr += local_cols * datum_size;
  }
  /* Process p-1 reads in the matrix one row at a time and
distributes each row among the MPI processes. */
if (id == (p-1))
buffer = my_malloc (id, *n * datum_size);
  create_mixed_xfer_arrays (id,p,*n,&send_count,&send_disp);
for (i = 0; i < *m; i++) {
if (id == (p-1))
fread (buffer, datum_size, *n, infileptr);
     MPI_Scatterv (buffer, send_count, send_disp, dtype,
       (*storage)+i*local_cols*datum_size, local_cols,
dtype, p-1, comm);
  }
free (send_count);
free (send_disp);
if (id == (p-1)) free (buffer);
}
/*
 *   Process p-1 opens a file and inputs a two-dimensional
 *   matrix, reading and distributing blocks of rows to the
 *   other processes.
 */
void read_row_striped_matrix (
char        *s,        /* IN - File name */
void      ***subs,     /* OUT - 2D submatrix indices */
void       **storage,  /* OUT - Submatrix stored here */
  MPI_Datatype dtype,    /* IN - Matrix element type */
int         *m,        /* OUT - Matrix rows */
int         *n,        /* OUT - Matrix cols */
  MPI_Comm     comm)     /* IN - Communicator */
{
int          datum_size;   /* Size of matrix element */
```

```
int         i;
int         id;          /* Process rank */
  FILE       *infileptr;    /* Input file pointer */
int         local_rows;   /* Rows on this proc */
void       **lptr;         /* Pointer into 'subs' */
int         p;           /* Number of processes */
void        *rptr;         /* Pointer into 'storage' */
  MPI_Status   status;       /* Result of receive */
int         x;           /* Result of read */
  MPI_Comm_size (comm, &p);
  MPI_Comm_rank (comm, &id);
  datum_size = get_size (dtype);
  /* Process p-1 opens file, reads size of matrix,
and broadcasts matrix dimensions to other procs */
if (id == (p-1)) {
infileptr = fopen (s, "r");
if (infileptr == NULL) *m = 0;
else {
fread (m, sizeof(int), 1, infileptr);
fread (n, sizeof(int), 1, infileptr);
     }
  }
  MPI_Bcast (m, 1, MPI_INT, p-1, comm);
if (!(*m))MPI_Abort (MPI_COMM_WORLD, OPEN_FILE_ERROR);
  MPI_Bcast (n, 1, MPI_INT, p-1, comm);
  local_rows = BLOCK_SIZE(id,p,*m);
  /* Dynamically allocate matrix. Allow double subscripting
through 'a'. */
  *storage = (void *) my_malloc (id,
      local_rows * *n * datum_size);
  *subs = (void **) my_malloc (id, local_rows * PTR_SIZE);
lptr = (void *) &(*subs[0]);
rptr = (void *) *storage;
```

```
for (i = 0; i < local_rows; i++) {
     *(lptr++)= (void *) rptr;
rptr += *n * datum_size;
  }

  /* Process p-1 reads blocks of rows from file and
sends each block to the correct destination process.
     The last block it keeps. */
if (id == (p-1)) {
for (i = 0; i < p-1; i++) {
        x = fread (*storage, datum_size,
          BLOCK_SIZE(i,p,*m) * *n, infileptr);
        MPI_Send (*storage, BLOCK_SIZE(i,p,*m) * *n, dtype,
i, DATA_MSG, comm);
     }
     x = fread (*storage, datum_size, local_rows * *n,
infileptr);
fclose (infileptr);
  } else
     MPI_Recv (*storage, local_rows * *n, dtype, p-1,
       DATA_MSG, comm, &status);
}
/*
 *   Open a file containing a vector, read its contents,
 *   and distributed the elements by block among the
 *   processes in a communicator.
 */
void read_block_vector (
char        *s,      /* IN - File name */
void        **v,     /* OUT - Subvector */
   MPI_Datatype dtype,  /* IN - Element type */
int         *n,      /* OUT - Vector length */
   MPI_Comm     comm)   /* IN - Communicator */
```

```
{
int        datum_size;   /* Bytes per element */
int        i;
  FILE      *infileptr;    /* Input file pointer */
int        local_els;    /* Elements on this proc */
  MPI_Status status;       /* Result of receive */
int        id;           /* Process rank */
int        p;            /* Number of processes */
int        x;            /* Result of read */
  datum_size = get_size (dtype);
  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &id);
  /* Process p-1 opens file, determines number of vector
elements, and broadcasts this value to the other
processes. */
if (id == (p-1)) {
infileptr = fopen (s, "r");
if (infileptr == NULL) *n = 0;
else fread (n, sizeof(int), 1, infileptr);
  }
  MPI_Bcast (n, 1, MPI_INT, p-1, comm);
if (! *n) {
if (!id) {
printf ("Input file '%s' cannot be opened\n", s);
fflush (stdout);
     }
  }
  /* Block mapping of vector elements to processes */
  local_els = BLOCK_SIZE(id,p,*n);
  /* Dynamically allocate vector. */

  *v = my_malloc (id, local_els * datum_size);
if (id == (p-1)) {
```

```
for (i = 0; i < p-1; i++) {
        x = fread (*v, datum_size, BLOCK_SIZE(i,p,*n),
infileptr);
        MPI_Send (*v, BLOCK_SIZE(i,p,*n), dtype, i, DATA_MSG,
comm);
      }
      x = fread (*v, datum_size, BLOCK_SIZE(id,p,*n),
infileptr);
fclose (infileptr);
   } else {
      MPI_Recv (*v, BLOCK_SIZE(id,p,*n), dtype, p-1, DATA_MSG,
comm, &status);
   }
}
/*   Open a file containing a vector, read its contents,
and replicate the vector among all processes in a
communicator. */
void read_replicated_vector (
char        *s,      /* IN - File name */
void        **v,      /* OUT - Vector */
   MPI_Datatype dtype,  /* IN - Vector type */
int         *n,      /* OUT - Vector length */
   MPI_Comm     comm)   /* IN - Communicator */
{
int        datum_size; /* Bytes per vector element */
int        i;
int        id;         /* Process rank */
   FILE      *infileptr;  /* Input file pointer */
int        p;          /* Number of processes */
   MPI_Comm_rank (comm, &id);
   MPI_Comm_size (comm, &p);
   datum_size = get_size (dtype);
if (id == (0)) { //Sahalu: 0 was p-1
```

```
infileptr = fopen (s, "r");
if (infileptr == NULL) *n = 0;
else fread (n, sizeof(int), 1, infileptr);
  }
  MPI_Bcast (n, 1, MPI_INT, 0, MPI_COMM_WORLD);  //Sahalu: 0 was p-1
if (! *n) terminate (id, "Cannot open vector file");
  *v = my_malloc (id, *n * datum_size);
if (id == (0)) { //Sahalu: 0 was p-1
fread (*v, datum_size, *n, infileptr);
fclose (infileptr);
  }
  MPI_Bcast (*v, *n, dtype, 0, MPI_COMM_WORLD); //Sahalu: 0 was p-1
}
/******************** OUTPUT FUNCTIONS ********************/
/*
 *   Print elements of a doubly-subscripted array.
 */
void print_submatrix (
void      **a,      /* OUT - Doubly-subscripted array */
  MPI_Datatype dtype,   /* OUT - Type of array elements */
int         rows,    /* OUT - Matrix rows */
int         cols)    /* OUT - Matrix cols */
{
int i, j;
for (i = 0; i < rows; i++) {
for (j = 0; j < cols; j++) {
if (dtype == MPI_DOUBLE)
printf ("%6.3f ", ((double **)a)[i][j]);
else {
if (dtype == MPI_FLOAT)
printf ("%6.3f ", ((float **)a)[i][j]);
else if (dtype == MPI_INT)
printf ("%6d ", ((int **)a)[i][j]);
```

```
        }
      }
putchar ('\n');
    }
}
/*
 *   Print elements of a singly-subscripted array.
 */
void print_subvector (
void        *a,       /* IN - Array pointer */
  MPI_Datatype dtype,   /* IN - Array type */
int         n)       /* IN - Array size */
{
int i;
for (i = 0; i < n; i++) {
if (dtype == MPI_DOUBLE)
printf ("%6.3f ", ((double *)a)[i]);
else {
if (dtype == MPI_FLOAT)
printf ("%6.3f ", ((float *)a)[i]);
else if (dtype == MPI_INT)
printf ("%6d ", ((int *)a)[i]);
      }
  }
}
/*
 *   Print a matrix distributed checkerboard fashion among
 *   the processes in a communicator.
 */
void print_checkerboard_matrix (
void       **a,           /* IN -2D matrix */
  MPI_Datatype dtype,        /* IN -Matrix element type */
int         m,           /* IN -Matrix rows */
```

```
int          n,           /* IN -Matrix columns */
  MPI_Comm     grid_comm)    /* IN - Communicator */
{
void      *buffer;         /* Room to hold 1 matrix row */
int       coords[2];      /* Grid coords of process
sending elements */
int       datum_size;     /* Bytes per matrix element */
int       els;            /* Elements received */
int       grid_coords[2]; /* Coords of this process */
int       grid_id;        /* Process rank in grid */
int       grid_period[2]; /* Wraparound */
int       grid_size[2];   /* Dims of process grid */
int       i, j, k;
void      *laddr;          /* Where to put subrow */
int       local_cols;     /* Matrix cols on this proc */
int       p;              /* Number of processes */
int       src;            /* ID of proc with subrow */
  MPI_Status status;         /* Result of receive */
  MPI_Comm_rank (grid_comm, &grid_id);
  MPI_Comm_size (grid_comm, &p);
  datum_size = get_size (dtype);
  MPI_Cart_get (grid_comm, 2, grid_size, grid_period,
    grid_coords);
  local_cols = BLOCK_SIZE(grid_coords[1],grid_size[1],n);
if (!grid_id)
buffer = my_malloc (grid_id, n*datum_size);
  /* For each row of the process grid */
for (i = 0; i < grid_size[0]; i++) {
coords[0] = i;
    /* For each matrix row controlled by the process row */
for (j = 0; j < BLOCK_SIZE(i,grid_size[0],m); j++) {
      /* Collect the matrix row on grid process 0 and
print it */
```

```
if (!grid_id) {
for (k = 0; k < grid_size[1]; k++) {
coords[1] = k;
            MPI_Cart_rank (grid_comm, coords, &src);
els = BLOCK_SIZE(k,grid_size[1],n);
laddr = buffer +
              BLOCK_LOW(k,grid_size[1],n) * datum_size;
if (src == 0) {
memcpy (laddr, a[j], els * datum_size);
            } else {
              MPI_Recv(laddr, els, dtype, src, 0,
                grid_comm, &status);
            }
          }
          print_subvector (buffer, dtype, n);
putchar ('\n');
        } else if (grid_coords[0] == i) {
          MPI_Send (a[j], local_cols, dtype, 0, 0,
            grid_comm);
        }
      }
  }
if (!grid_id) {
free (buffer);
putchar ('\n');
  }
}
/*
 *   Print a matrix that has a columnwise-block-striped data
 *   decomposition among the elements of a communicator.
 */
void print_col_striped_matrix (
void       **a,       /* IN - 2D array */
```

```
  MPI_Datatype dtype,   /* IN - Type of matrix elements */
int          m,       /* IN - Matrix rows */
int          n,       /* IN - Matrix cols */
  MPI_Comm     comm)    /* IN - Communicator */
{
  MPI_Status status;     /* Result of receive */
int       datum_size; /* Bytes per matrix element */
void      *buffer;     /* Enough room to hold 1 row */
int       i, j;
int       id;          /* Process rank */
int       p;           /* Number of processes */
int*      rec_count;  /* Elements received per proc */
int*      rec_disp;   /* Offset of each proc's block */
  MPI_Comm_rank (comm, &id);
  MPI_Comm_size (comm, &p);
  datum_size = get_size (dtype);
  create_mixed_xfer_arrays (id, p, n, &rec_count,&rec_disp);
if (!id)
buffer = my_malloc (id, n*datum_size);
for (i = 0; i < m; i++) {
     MPI_Gatherv (a[i], BLOCK_SIZE(id,p,n), dtype, buffer,
       rec_count, rec_disp, dtype, 0, MPI_COMM_WORLD);
if (!id) {
       print_subvector (buffer, dtype, n);
putchar ('\n');
     }
  }
free (rec_count);
free (rec_disp);
if (!id) {
free (buffer);
putchar ('\n');
  }
```

```
}
/*
 *   Print a matrix that is distributed in row-striped
 *   fashion among the processes in a communicator.
 */
void print_row_striped_matrix (
void **a,           /* IN - 2D array */
  MPI_Datatype dtype,  /* IN - Matrix element type */
int m,              /* IN - Matrix rows */
int n,              /* IN - Matrix cols */
  MPI_Comm comm)      /* IN - Communicator */
{
  MPI_Status  status;         /* Result of receive */
void      *bstorage;       /* Elements received from
another process */
void      **b;              /* 2D array indexing into
                            'bstorage' */
int       datum_size;      /* Bytes per element */
int       i;
int       id;             /* Process rank */
int       local_rows;      /* This proc's rows */
int       max_block_size;  /* Most matrix rows held by
any process */
int       prompt;          /* Dummy variable */
int       p;              /* Number of processes */
  MPI_Comm_rank (comm, &id);
  MPI_Comm_size (comm, &p);
  local_rows = BLOCK_SIZE(id,p,m);
if (!id) {
    print_submatrix (a, dtype, local_rows, n);
if (p > 1) {
       datum_size = get_size (dtype);
       max_block_size = BLOCK_SIZE(p-1,p,m);
```

```
bstorage = my_malloc (id,
           max_block_size * n * datum_size);
         b = (void **) my_malloc (id,
           max_block_size * datum_size);
b[0] = bstorage;
for (i = 1; i < max_block_size; i++) {
b[i] = b[i-1] + n * datum_size;
         }
for (i = 1; i < p; i++) {
           MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG,
             MPI_COMM_WORLD);
           MPI_Recv (bstorage, BLOCK_SIZE(i,p,m)*n, dtype,
i, RESPONSE_MSG, MPI_COMM_WORLD, &status);
           print_submatrix (b, dtype, BLOCK_SIZE(i,p,m), n);
         }
free (b);
free (bstorage);
      }
putchar ('\n');
   } else {
      MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG,
        MPI_COMM_WORLD, &status);
      MPI_Send (*a, local_rows * n, dtype, 0, RESPONSE_MSG,
        MPI_COMM_WORLD);
   }
}
/*
 *   Print a vector that is block distributed among the
 *   processes in a communicator.
 */
void print_block_vector (
void        *v,       /* IN - Address of vector */
   MPI_Datatype dtype,   /* IN - Vector element type */
```

```
int          n,       /* IN - Elements in vector */
  MPI_Comm     comm)    /* IN - Communicator */
{
int       datum_size; /* Bytes per vector element */
int       i;
int       prompt;     /* Dummy variable */
  MPI_Status status;     /* Result of receive */
void      *tmp;       /* Other process's subvector */
int       id;         /* Process rank */
int       p;          /* Number of processes */
  MPI_Comm_size (comm, &p);
  MPI_Comm_rank (comm, &id);
  datum_size = get_size (dtype);
if (!id) {
    print_subvector (v, dtype, BLOCK_SIZE(id,p,n));
if (p > 1) {
tmp = my_malloc (id,BLOCK_SIZE(p-1,p,n)*datum_size);
for (i = 1; i < p; i++) {
          MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG,
comm);
          MPI_Recv (tmp, BLOCK_SIZE(i,p,n), dtype, i,
            RESPONSE_MSG, comm, &status);
          print_subvector (tmp, dtype, BLOCK_SIZE(i,p,n));
        }
free (tmp);
    }
printf ("\n\n");
  } else {
    MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG, comm,
&status);
    MPI_Send (v, BLOCK_SIZE(id,p,n), dtype, 0,
      RESPONSE_MSG, comm);
  }
```

```
}



/*
 *   Print a vector that is replicated among the processes
 *   in a communicator.
 */
void print_replicated_vector (
void        *v,      /* IN - Address of vector */
  MPI_Datatype dtype,  /* IN - Vector element type */
int          n,      /* IN - Elements in vector */
  MPI_Comm     comm)   /* IN - Communicator */
{
int id;               /* Process rank */
  MPI_Comm_rank (comm, &id);
if (!id) {
     print_subvector (v, dtype, n);
printf ("\n\n");
  }
}
```

APPENDIX C

Windows Batch Scrips

```
@echo off

REM receives the dimension of the array as input arguments and generates two text files(arrays and vector ) with integer arrays based on the dimension

REM the loop with variable i is used to pass the different sizes of arrays and vectors already generated for the program

REM the loop with variable K is to run the mpi program Kth time so that at the end the average speed can be calculated


for %%I in (4000) do (

		for /l %%K in (1, 1, 10) do (

			mpiexec -hosts 4 charley-pc cs-pc FAITHAKOR-HP user-c3e3057e86 mv1 array_int_%%I.txt vector_int_%%I.txt 1

			mpiexec -hosts 4 charley-pc cs-pc FAITHAKOR-HP user-c3e3057e86 mv2 array_int_%%I.txt vector_int_%%I.txt 1

			mpiexec -hosts 4 charley-pc cs-pc FAITHAKOR-HP user-c3e3057e86 mv3 array_int_%%I.txt vector_int_%%I.txt 1

			mpiexec -hosts 4 charley-pc cs-pc FAITHAKOR-HP user-c3e3057e86 mv4 array_int_2000.txt array_int_2000.txt 1

			echo .

		)

	echo next dimension

)


@echo off

REM receives the dimension of the array as input arguments and generates two text files(arrays and vector ) with integer arrays based on the dimension

REM the loop with variable i is used to pass the different sizes of arrays and vectors already generated for the program

REM the loop with variable J is used to pass the number of process each host is to launch. all hosts launch equal number of processes

REM the loop with variable K is run the mpi program Kth time so that at the end the average speed can be calculated


for %%I in (4000) do (
```

```
for /l %%J in (17, 1, 20) DO (
        for /l %%K in (1, 1, 3) do (
mpiexec -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J mv1_m array_int_%%I.txt vector_int_%%I.txt %%J
mpiexec -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J mv2_m array_int_%%I.txt vector_int_%%I.txt %%J
mpiexec -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J mv3_m array_int_%%I.txt vector_int_%%I.txt %%J
mpiexec -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J mv4_m array_int_2000.txt array_int_2000.txt %%J
        echo .
        )
mpiexec -log -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J  mv1_m array_int_%%I.txt vector_int_%%I.txt %%J
mpiexec -log -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J  mv2_m array_int_%%I.txt vector_int_%%I.txt %%J
mpiexec -log -timeout 420 -hosts 2 charley-pc %%J cs-pc %%J  mv3_m array_int_%%I.txt vector_int_%%I.txt %%J
mpiexec -log -timeout 600 -hosts 2 charley-pc %%J cs-pc %%J  mv4_mc array_int_2000.txt array_int_2000.txt %%J
        echo .
        echo next process number
)
echo next array and vector
)
```