

OPTIMIZING MULTI-DIMENSIONAL MPI
COMMUNICATIONS ON MULTI-CORE
ARCHITECTURES

by
Christer Karlsson

© Copyright by Christer Karlsson, 2012

All Rights Reserved

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Mathematical and Computer Sciences).

Golden, Colorado

Date _____

Signed: _____

Christer Karlsson

Signed: _____

Dr. Zizhong Chen
Thesis Advisor

Golden, Colorado

Date _____

Signed: _____

Dr. Randy Haupt
Professor and Head
Department of Electrical Engineering and Computer Science

ABSTRACT

In today's high performance computing, many Message Passing Interface (MPI) programs (e.g., ScaLAPACK applications, High Performance Linpack Benchmark (HPL), and most PDE solvers based on domain decomposition methods) organize their computational processes as multidimensional Cartesian grids. Applications often need to communicate in every dimension of the Cartesian grid. While extensive optimizations have been performed on single dimensional communications such as the standard MPI collective communications, little work has been done to optimize multidimensional communications. We study the impact of the MPI process-to-core mapping on the performance of multidimensional MPI communications on Cartesian grid. While the default process-to-core mappings in today's state-of-the-art MPI implementations are often optimal for single dimensional communications, we show that they are often sub-optimal for multidimensional communications. We propose an application-level multicore-aware process-to-core re-mapping scheme that is capable of achieving optimal performance for multidimensional communication operations. The application-level solution does not require any changes to the MPI's implementations; the optimization will occur in the application layer. Experiments demonstrate that a multicore-aware process-to-core re-mapping scheme improves the performance of multidimensional MPI communications by up to 80% over the default mapping scheme on the world's current third fastest supercomputer, Jaguar, located at the Oak Ridge National Laboratory.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES	viii
LIST OF TABLES	xi
LIST OF SYMBOLS	xiii
ACKNOWLEDGMENTS	xiv
DEDICATION	xv
CHAPTER 1 INTRODUCTION	1
1.1 Dissertation Organization and Research Contribution	4
CHAPTER 2 BACKGROUND	7
2.1 Collective MPI communications	7
2.2 Related work	11
2.2.1 Memory Usage	11
2.2.2 Optimizing communication on System level	13
2.2.3 Optimizing the MPI methods	14
2.2.4 Multidimensional Communication & MPI	15
2.2.5 Optimizing multidimensional neighbor communication	15
CHAPTER 3 THE EXTENDED PARALLEL PING-PONG TEST	17
3.1 Performance Measuring and Benchmarking	17
3.2 Latency Test	20
3.3 Bandwidth Tests	22

3.4	Bidirectional Bandwidth Test	22
3.5	Latency Test Multiple Pairs	22
3.6	Bandwidth Test Multiple Pairs	26
3.7	Evaluation	26
3.7.1	System Specifications	26
3.7.2	Latency and Bandwidth Tests	28
3.7.3	Latency and Bandwidth Tests Multiple Pairs	32
3.7.4	The BlueDrop System	34
CHAPTER 4 SINGLE NODE OPTIMIZATION		37
4.1	Communication on Shared Memory Parallel Node	37
4.2	Fastest Edge First	43
4.3	Channel Aware Ordering	44
4.4	Random Ordering	48
4.5	Re-Mapping	48
4.6	Evaluation	56
4.6.1	Theoretical Assumptions	57
4.6.2	Experiments and Results	57
CHAPTER 5 MULTI-DIMENSIONAL MPI COMMUNICATIONS		62
5.1	Background	62
5.2	Tiling	67
5.2.1	Node mapping	69
5.3	Theoretical Analysis	70
5.3.1	Multi-Dimensional MPI Communication Based on Pipeline Algorithm	71

5.3.2	Two-Dimensional Case	72
5.3.3	d -Dimensional Matrix Arrays	77
5.3.4	Multi-Dimensional MPI Communications Based on Binomial Tree Algorithm	79
5.4	Evaluation	82
5.4.1	<i>Alamode</i> Results	83
5.4.2	<i>RA</i> Results	84
5.4.3	<i>Jaguar</i> Results	88
5.4.4	Combined Tables for the Two-Dimensional Results on all Systems	91
5.4.5	Three-Dimensional Results on <i>RA</i> and <i>Jaguar</i>	91
5.4.6	Re-mapping Overhead	91
CHAPTER 6 APPLICATION IMPLEMENTATION		96
6.1	Matrix-Matrix Multiplication	96
6.1.1	Results	101
6.1.2	Overhead	102
6.2	The N -body Problem	103
6.2.1	Communication	106
6.2.2	Results	109
CHAPTER 7 CONCLUSION		111
7.1	Summary	111
7.2	Future Work	112
REFERENCES CITED		113
APPENDIX - DESCRIPTION OF MPI-FUNCTIONS		123

A.1	Point-to-Point Communication	123
A.1.1	Blocking Communication	123
A.2	Non-blocking Communication	126
A.3	Collective Communications	129
A.3.1	All-To-One Communication	129
A.3.2	One-To-All Communication	132
A.3.3	All-To-All Communication	133
A.3.4	Other	136
A.4	MPI Groups and Communicators	137
A.4.1	Groups	138
A.4.2	Communicators	139

LIST OF FIGURES

Figure 1.1	An example of an HPC system	2
Figure 1.2	Example of a 2D communication grid	4
Figure 3.1	Multiple Pairs Parallel Ping-Pong Test	24
Figure 3.2	Linear Regression <i>Alamode</i> , both <i>Inter-</i> and <i>Intra-</i> Communication.	29
Figure 3.3	Linear Regression <i>RA</i> , both <i>Inter-</i> and <i>Intra-</i> Communication.	30
Figure 3.4	Linear Regression <i>Jaguar</i> , both <i>Inter-</i> and <i>Intra-</i> Communication.	31
Figure 4.1	Example of a modern multicore node	38
Figure 4.2	Example of a Power7 Node with four Processors.	39
Figure 4.3	Example of a binomial tree	40
Figure 4.4	Example of a binomial tree	42
Figure 4.5	Example of an FEF ordering with two quad-core processors.	44
Figure 4.6	Example of Extended Parallel Ping-Pong Tests	47
Figure 4.7	Example of an CAO ordering with two quad-core processors.	48
Figure 4.8	Example of a Process to Core Mapping	51
Figure 4.9	Example of two different Re-mappings; $n = 4$, $k = 4$	52
Figure 4.10	Example of the rank of single node on the RA cluster	53
Figure 4.11	Re-mapping of the original Rank on RA	54
Figure 5.1	An example of a 3 dimensional process grid.	63
Figure 5.2	Example of 2D <code>MPI_Bcast()</code> , two sequential broadcasts using two different communicators.	64

Figure 5.3	An example of a PDE: 1000x1000 grid-elements solved on 100 processes. . .	65
Figure 5.4	An example of the Inter vs. Intra Communication on small grids using four nodes with four cores each.	68
Figure 5.5	An example of inter-node messages on a twelve core node using different tiles.	69
Figure 5.6	Example of a 2D mappings with $n = 4$ and $k = 4$	70
Figure 5.7	Reduction function using the Pipeline Algorithm	72
Figure 5.8	Matrix reduction in 2 dimensions	73
Figure 5.9	Example of a 2D binomial tree communication grid	80
Figure 5.10	Example of a non-tiled 2D binomial tree communication grid	80
Figure 5.11	Example of a tiled 2D binomial tree communication grid	81
Figure 5.12	Two Dimensional MPI_Sctr() on the <i>Alamode</i> System	83
Figure 5.13	Two Dimensional MPI_Gthr() on the <i>Alamode</i> System	84
Figure 5.14	<i>Broadcast</i> Functions on the <i>Alamode</i> System ($m = 8 MB$).	85
Figure 5.15	<i>Broadcast</i> Functions on the <i>RA</i> Cluster ($m = 16 MB$).	86
Figure 5.16	Two Dimensional MPI_Sctr() on the <i>RA</i> Cluster	87
Figure 5.17	Two Dimensional MPI_Gthr() on the <i>RA</i> Cluster	87
Figure 5.18	<i>Broadcast</i> Functions on the <i>Jaguar</i> Cluster ($m = 64 MB$).	89
Figure 5.19	Two Dimensional MPI_Sctr() on the <i>Jaguar</i> Cluster	90
Figure 5.20	Two Dimensional MPI_Gthr() on the <i>Jaguar</i> Cluster	90
Figure 5.21	Throughput values for MPI_Bcst() on the <i>Jaguar</i> Cluster	90
Figure 6.1	Matrix-Matrix Multiplication with a one dimension decomposition	97
Figure 6.2	Matrix-Matrix Multiplication with a two dimension decomposition	98

Figure 6.3	Example of one step in of the Parallel Matrix-Matrix multiplication Algorithm.	100
Figure 6.4	N -body problem with four particles	104
Figure 6.5	The <code>MPI_Gather()</code> communication	106
Figure 6.6	The <code>MPI_Allgather()</code> communication	106
Figure 6.7	Two different implementations of <code>MPI_Allgather()</code> communication. . .	108
Figure A.1	Example of <code>MPI_GATHER()</code>	130
Figure A.2	Example of <code>MPI_REDUCE()</code>	131
Figure A.3	Example of <code>MPI_SCATTER()</code>	132
Figure A.4	Example of <code>MPI_BCAST()</code>	133
Figure A.5	Example of <code>MPI_ALLGATHER()</code>	134
Figure A.6	Example of <code>MPI_ALLTOALL()</code>	135
Figure A.7	Example of <code>MPI_ALLREDUCE()</code>	136
Figure A.8	Example of <code>MPI_REDUCE_SCATTER()</code>	136
Figure A.9	Example of <code>MPI_SCAN()</code>	137

LIST OF TABLES

Table 3.1	Network Characteristics <i>RA-cluster</i> (InfiniBand Cisco 7024 IB server Switch).	18
Table 3.2	Network Characteristics <i>Alamode-lab</i> (Fast Ethernet (1Gbps) switched network links).	18
Table 3.3	Example of a Round-Trip Time Matrix <i>Alamode-lab</i>	21
Table 3.4	The combined results from the Linear Regressions.	28
Table 3.5	Latency and Bandwidth Tests Multiple Pairs on <i>Alamode m = 8 MB</i>	32
Table 3.6	Latency and Bandwidth Tests Multiple Pairs on <i>RA m = 8 MB</i>	32
Table 3.7	Latency and Bandwidth Tests Multiple Pairs on <i>Jaguar m = 8 MB</i>	32
Table 3.8	Total Bandwidth Multiple Pairs on <i>Alamode m = 8 MB</i>	33
Table 3.9	Total Bandwidth Multiple Pairs on <i>RA m = 8 MB</i>	33
Table 3.10	Total Bandwidth Multiple Pairs on <i>Jaguar m = 8 MB</i>	34
Table 3.11	The combined results from the Multiple Pair test.	34
Table 3.12	RTT for Ping-Pong test on <i>BlueDrop (m = 512 kB)</i>	35
Table 3.13	The RTT for Multi-pair bandwidth test between processes in the same group (= 512 kB).	35
Table 3.14	The RTT for Multi pair bandwidth test between processes in different groups (= 512 kB).	36
Table 4.1	An example of a Cost Matrix (round trip travel-times) from the RA supercomputing cluster at the School of Mines using 8 cores and a scan size of 16 kB (μs).	38
Table 4.2	An example of a Cost Matrix (round trip travel-times) from the Power7 node using 32 cores and a scan size of 16 kB (μs).	38

Table 4.3	A message of 8 m is dispersed to eight processes.	41
Table 4.4	A message of 8 m , scattered using the <i>FFE</i> -communicator.	55
Table 4.5	A message of 8 m , scattered using the <i>FFE</i> -communicator.	56
Table 4.6	The cost to scatter a message of size $8m$ to eight cores in two groups, allowing two messages to be sent in parallel.	57
Table 4.7	Cost Matrix results (round trip travel-times) for all three systems.	58
Table 4.8	Extended Ping-Pong results for all three systems.	58
Table 4.9	The results from RA, $n = 20$ and 10 MB message size.	59
Table 4.10	The results from Kraken, $n = 20$ and 10 MB message size.	60
Table 4.11	The results from BlueDrop, $n = 20$ and 10 MB message size.	60
Table 4.12	The step by step results from BlueDrop, $n = 20$ and 10 MB message size. . .	61
Table 5.1	<code>MPI_Bcast()</code> Tests on <i>RA</i> and <i>Jaguar</i> Implementing Three-Dimensional Communication Grids (16 MB message size).	92
Table 5.2	Two Dimensional Tests on <i>Alamode</i> (8 MB Message Size).	93
Table 5.3	Two Dimensional Tests on <i>RA</i> (16 MB Message Size).	94
Table 5.4	Two Dimensional Tests on <i>Jaguar</i> (64 MB Message Size).	95
Table 6.1	Matrix-Matrix Multiplication, 2048×2048 sub-matrix size	101
Table 6.2	Tiling function overhead with Matrix-Matrix Multiplication	102
Table 6.3	Results from the N -body tests, $N/p = 250$ <i>time steps</i> = 1000 and $\Delta t = 0.01$	110
Table A.1	The basic MPI datatypes and the corresponding C types.	124
Table A.2	The basic reduction operation that are predefined in MPI.	131
Table A.3	Example of the arguments for a <code>MPI_COMM_SPLIT</code> call on twelve elements. . .	141

LIST OF SYMBOLS

Bandwidth	β
External Bandwidth (<i>Inter-communication</i>)	β_E
Internal Bandwidth (<i>Intra-communication</i>)	β_I
Number of cores per node	k
Number of cores per node in i^{th} dimension	k_i
Number of dimensions	d
Initiation cost (latency)	α
External Latency (<i>Inter-communication</i>)	α_E
Internal Latency (<i>Intra-communication</i>)	α_I
Number of nodes	n
Number of nodes in i^{th} dimension	n_i
Mapping vector	\mathbb{M}
Message size	m
Average message size	\bar{m}
Process i	$p[i]$
Number of MPI Processes	p
Number of MPI Processes in i^{th} dimension	p_i
A process position in a given communicator	<i>rank</i>
The array of the positions for all processes in a given communicator	rank
Time cost for process i to communicate to process j	$T_{i,j}(m)$

ACKNOWLEDGMENTS

This work was made possible through the support and encouragement of so many. I would like to acknowledge and thank my advisor, Dr. Zizhong Chen and all of the students in the Supercomputing Laboratory Group at Colorado School of Mines. Your support, help, feedback and encouragement through this process has been invaluable, and the grants from National Science Foundation and Department of Energy (CCF-1118039, OCI-1150273, CNS-1118037 and DE-FE-0000988) made much of this possible. Also, Dr. Catherine Skokan at the Department of Electrical Engineering and Computer Science for her mentoring, support and funding, which have been the solid foundation needed to ensure the success of this path.

I also like to extend my acknowledgment to all the professors and teachers that have over the years in high school and college influenced me and helped me become who I am. I cannot, of course, mention them all, but some of the most influential have been: Drs. Diana and William Spears and Dr. Bryan Shader at the University of Wyoming; your enthusiasm and dedication to both teaching and research has been an inspiration and you have over the years gone from advisor to mentor to true friends. Without your support and help I might never have embarked on or finished this journey.

Last, but probably most important, I would like to acknowledge my family and friends; you have always been there, always supported me and as the Swedish proverb says: “. . . when I least deserve it, that is when I need it the most.”

Till minnet av min mamma, som alltid var där för mig, men aldrig fick chansen att se den färdiga produkten.

In memory of my mother, who was always there for me but never had the chance to see the finished product.

CHAPTER 1

INTRODUCTION

The Message Passing Interface (*MPI*) implementations that many programmers continue to use were originally not designed to account for the necessity of multi-core architectures. The preference for the multi-core approach to High-Performance Computing (*HPC*) was essential to surpass the processing limits imposed by the power- and thermic-barriers. As such, today's machines are no longer the homogeneous systems for which the *MPI* processes were originally developed. Because of the unquenchable demand of computing power from the domain sciences motivates deployment of ever more powerful *HPC* systems, multicore clusters have become the foremost form of *HPC* systems, and exhibit a rapid increase in the number of cores per node. The top ranking machine in the latest Top500 list, the Sequoia at the Department of Energys Lawrence Livermore National Laboratory, uses more than one and a half a million cores [1, 2]. Processors with 12 cores are available from major vendors and it is common to have these deployed in multiple socket boards featuring 8 to 48 cores with network-style interconnection between caches or to the memory banks e.g. Intel QPI or AMD Hyper-transport (Figure 1.1).

In the last few years, multi-cores nodes have effectively replaced single processor nodes in almost all commodity hardware based computing clusters due to their low cost and increased performance [3, 4]. This trend is expected to continue into the future as more cores are added to individual processors and the price comes down. Unfortunately, optimization methods for single-core processors do not directly translate to multi-core processors, as multi-core processors introduce new memory management and communication schemes for the end-user to deal with [5]. One issue that remains unsolved is how to improve performance for collective communication routines on multi-core clusters. This is a major issue because the communication models that underlie most communication routines are no longer completely

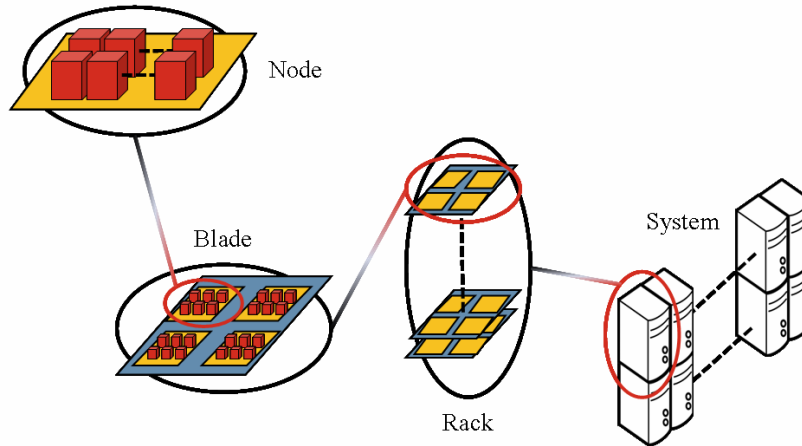


Figure 1.1: An example of an HPC system with racks, blades, nodes and cores

valid. This is due to a large disparity in communication speeds between processes on the same multi-core node and communication between nodes. This disparity effectively creates a two-tiered, heterogeneous network topology. Therefore, previous communication routines that assumed network homogeneity are no longer completely relevant. Further compounding the issue, many multi-core clusters do not have multiple network connections per node, which differentiates them from other cluster types such as SMP clusters [5]. In many instances, all processes on a single multi-core node will share a single network component as their connection to the network. Obviously, this creates a substantial problem as processes on the same node may be forced to wait to send or receive messages. The problem is further complicated by the fact that many of the machines today have multiple processors with multiple cores on a single node. The way these cores and processors share memory creates a non-homogeneous (latency and bandwidth is different depending which cores communicate with each other) communication as far down as the node level.

It has therefore become a critical difference between *Inter-* (communication between members in two different groups) and *Intra-*communications (communication between members within the same group) that MPI were not originally designed to handle. This affects the performance of HPC applications and becomes ever more noticeable as the numbers

of cores increase. Systems such as *Kraken* or *Jaguar* with their hundreds of thousands of cores would benefit greatly from having their collective communications optimized [6–8]. Recently, the optimization of MPI communications based on multi-core architectures have been extensively studied [9–11]. While this research has significantly improved the performance of many MPI operations, most of this research focuses on standard MPI communications where processes are organized as a one-dimensional array. However, in today’s high performance scientific computing, many MPI programs (e.g., ScaLAPACK applications, High Performance Linpack Benchmark HPL, and many PDE solvers based on domain decomposition methods) organize their computational processes as multidimensional process grids [12], with the communications often being performed in each dimension simultaneously. It’s these multidimensional communications that have not yet been addressed for optimization.

For example, in the widely used ScaLAPACK matrix-matrix multiplication kernel [13], computational processes are organized into a two-dimensional process grid. At the k^{th} iteration of the algorithm, the k^{th} row block has to be broadcast to all other processes in the same column and the k^{th} column block has to be broadcast to all other processes in the same row (see Figure 1.2). This way to communicate lacks a standard MPI solution and it is heavily dependent on the hardware topology and overall physical configuration. It is commonly accepted that cores, which are physically close to each other in the network grid, have a shorter communication time. Much time and effort has been spent to ensure that these neighboring cores communicate well together. However, most of the resulting improvements have been done assuming that all the involved cores always communicate as a group.

Little-to-no research has been done regarding the overall impact created when different sub-group configurations need to communicate together. Returning to our ScaLAPACK example, when the communicator was initialized all the cores were arranged to create a rank order, but this is often done with little regard to the communication pattern of the application. Depending on the rank ordering, communication along the row or column is optimized, but never both at the same time. The lack of optimization regarding these

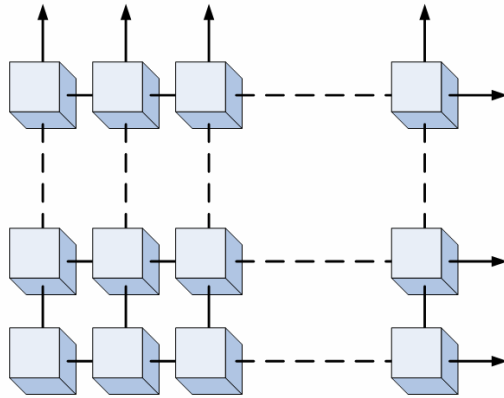


Figure 1.2: Example of a 2D communication grid; each process belongs to both a row- and a column-communicator

multidimensional communications often has a significant impact on the performance of the whole application.

This dissertation explores the impact of the MPI process-to-core mapping on the performance of communication operations of heterogeneous systems. We will use unmodified standard MPI communicators, but we will implement application-level redistribution of the process-to-core ordering to achieve optimal performance. We demonstrate that the default process-to-core mappings in today’s state-of-the-art MPI implementations provided by `MPI_Init()` and `MPI_Cart_create()` are often sub-optimal for multidimensional communications. In this study we are only looking at the node architecture because this is the first step in a larger undertaking. We will not address any system level impact such as closeness between nodes, amount of switches, or other hardware topology issues; however, the work will later be extended to address these issues.

1.1 Dissertation Organization and Research Contribution

Chapter 1: (this chapter) gives a short description of the problem and the background for our research.

Chapter 2: presents some necessary background information that will be helpful for the remainder of the dissertation. We also present and discuss some work related to this research.

Chapter 3: introduces our new and innovated approach to an extended ping-pong test with the purpose of determining the raw characteristics of the network. We have extended the standard ping-pong test with two new functions to help give us more information about the network. These extended tests provide information about the network characteristics both internally between pairs of cores on the same node, and externally between pairs of cores on different nodes. The results from these tests provide the foundation for the other chapters and vital information for the application-level redistribution of the process-to-core ordering.

Chapter 4: presents our work on re-mapping algorithm on the lowest level, on a single node. We are implementing and extending work done on non-homogeneous clusters into working algorithms for a single node using unmodified standard MPI communicators, but applying a runtime application-level multicore-aware process-to-core re-mapping scheme that is capable of achieving optimal performance for communication on binomial trees. The results demonstrate that the default process-to-core mappings in today’s state-of-the-art MPI implementations provided by `MPI_Init()` and `MPI_Cart_create()` are often sub-optimal.

Chapter 5: introduces a tile-based process-to-core re-mapping that is able to improve the performance of a d -dimensional broadcast by a factor of $\Theta(\frac{1}{d}ck^{\frac{d-1}{d}})$ over the default mapping, where c is the number of the network ports per node and k is the number of the computational cores per node. Multidimensional reduction, scatter and gather operations are also studied and similar results are obtained. Experimental results,

as demonstrated on the world's current sixth fastest supercomputer, *Jaguar*, at the Oak Ridge National Laboratory show that the proposed tile-based process-to-core re-mapping improves the performance of multidimensional communications by up to 80%.

Chapter 6: shows the work on two different implementation of the algorithms to improve the multidimensional communication. The first application is the Matrix-Matrix multiplication. We use a standard algorithm with `MPI_Bcast()` and `dgemm()` to multiply two large matrices. The changes in the multidimensional communication improved the performance by as much as 25%. In the second application we introduced our re-mapping algorithm into a n -body solver. A standard algorithm with an *All-to-All* communication and Newton's Gravitational Force Law to solve the simulation was used. Our results shows that not all applications, and not type of multidimensional communications are improved by our re-mapping algorithm.

CHAPTER 2

BACKGROUND

In this chapter, we present some background information that is necessary for the remainder of the dissertation. We also present and discuss some work related to this research.

2.1 Collective MPI communications

Message Passing Interface (MPI) has, since its completion and introduction in June of 1994, become widely accepted and used as the most common method for programming distributed-memory Multiple-instruction Multiple-Data (MIMD) systems. MPI is not a new programming language. It is a collection of functions and macros, or a library that can be utilized within other programming languages including *C*, *C++*, *FORTRAN* and any language that is able to interface such libraries, e.g. *C#*, *Java* and *Python*. MPI is portable and it supports the important aspect scalability through several of its design features [14]. As an illustration, an application can create subgroups of processes that, in turn, allows collective communication operations to limit their scope to the processes involved [10, 15–17]. Another technique used is to provide functionality without a computation that scales as the number of processes. For example, a two-dimensional Cartesian topology can be subdivided into its one-dimensional rows or columns without explicitly enumerating the processes [18–20] (see Appendix A.4.2).

MPI also defines a known, minimum behavior of message-passing implementations. This relieves the programmer from having to worry about certain problems that can arise. One example is that MPI guarantees that the underlying transmission of messages is reliable. The user needs not check if a message is received correctly.

Our research focuses on MPI's ability to create subgroups that allow collective communication operations with limited scope. Our intention is not to make any changes to

the functionality of MPI, nor to any of its algorithms. We strictly focus on application-level multicore-aware process-to-core re-mapping schemes that take into account not only the system hardware-configurations, but also the communications patterns of the processes involved.

The following is a shorter description of the MPI collective communication. For a more in depth description see A. The *MPI Collective Operations* involve communication among all processes in a process group (which could mean all processes or just a defined subset) and are often categorized into four different groups [21]:

- All-to-One
- One-to-All
- All-to-All
- Other

All-to-One are the MPI functions where a message is passed from all processes to a single root. This group contains the functions:

- `MPI_Gather()`
- `MPI_Gatherv()`
- `MPI_Reduce()`

In `MPI_Gather()` each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.

`MPI_Gatherv()` extends the functionality of `MPI_Gather()` by allowing a varying count of data from each process, by implementing the size of each message as an array. This allows for more flexibility as to where the data is placed on the root.

`MPI_Reduce()` combines the elements provided in the input buffer of each process in the group, using some operation, and stores the combined value in an output buffer of the process with rank root.

One-to-All are the MPI functions where a message is passed from a single process to all the other processes. This group contains the functions:

- `MPI_Scatter()`
- `MPI_Scatterv()`
- `MPI_Bcast()`

In `MPI_Bcast()` broadcasts a message from the process with rank root to all processes of the group.

`MPI_Scatter()` is the inverse operation to `MPI_Gather()`, a message from the root is split into n equal segments, the i th segment is sent to the i th process in the group.

`MPI_Scatterv()` is the inverse operation to `MPI_Gatherv()`.

All-to-All are the MPI functions where a message is passed from all processes to all the other processes. This can be done by a combination of an *All-to-One* followed by *One-to-All*. This group contains the functions:

- `MPI_Allgather()`
- `MPI_Alltoall()`
- `MPI_Allreduce()`
- `MPI_Reduce_scatter()`

In `MPI_Allgather()` can be thought of as `MPI_Gather()`, except all processes receive the result instead of just the root. The j th block of data sent from each process is received by every process and placed in the j th block of the buffer on each process.

`MPI_Alltoall()` is an extension of `MPI_Allgather()` to the case where each process sends distinct data to each of the receivers. The j th block sent from process i is received by process j and is placed in the i th block of the local buffer.

`MPI_Allreduce()` is the same as `MPI_Reduce()` except that the result appears in the receive buffer of all the group members.

`MPI_Reduce_scatter()` is the same as `MPI_Allreduce()` with the exception that the buffer is an array, and each element in the array is first reduced before it is scattered to all the processes.

Other are those MPI functions that do not fit in any of the previous categories. This group contains the functions:

- `MPI_Barrier()`
- `MPI_Scan()`

`MPI_Barrier()` blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

`MPI_Scan()` is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank i , the reduction of the values in the send buffers of processes with ranks $0, \dots, i$ (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for `MPI_Reduce()`.

The *Collective communications* are often characterized by:

- Involve coordinated communication within a group of processes identified by an MPI communicator
- Substitute for a more complex sequence of point-to-point calls

- All routines block until they are locally complete
- Communications may or may not be synchronized (implementation dependent)

In the multidimensional case it means that data is passed either to or from all members in each dimension (*row* or *column* etc.).

2.2 Related work

Much research has already been done with *point-to-point communication* and many researchers have taken steps to improve individual pieces of the collective communication process. These improvements include items such as improved memory usage or further optimizing MPI and system level functions. These separate improvements subsequently benefit our own application-level solution.

2.2.1 Memory Usage

The power- and thermic-barriers may limit processor speed, but improvements regarding memory will naturally increase the overall performance of the processes. Research such as this is being conducted to develop a more efficient method of memory usage in the multicore design in an attempt to further surpass the power- and thermic-barriers and further optimize the entire system [9, 22, 23]. They have designed four sets of experiments to study: latency and bandwidth, message distribution, potential bottleneck identification, and scalability tests.

- Latency and Bandwidth: These are the standard Ping-Pong latency and bandwidth tests to characterize the three levels of communication in a multi-core cluster: intra-CMP, inter-CMP, and inter-node communication.
- Message Distribution: This defines the message distribution as a two dimensional metric. One dimension is with respect to the communication channel, i.e. the percentage of traffic going through intra-CMP, inter-CMP, and inter-node respectively. The other

dimension is in terms of message size. This experiment is very important because understanding message distribution facilitates communication middleware developers, e.g. MPI implementors, to optimize critical communication channels and message size range for applications. The message distribution is measured in terms of both number of messages and data volume.

- **Potential Bottleneck Identification:** In this experiment, they run application level benchmarks on different configurations, e.g. four processes on the same node, four processes on two different nodes, and four processes on four different nodes. The benchmark target is to discover any potential bottlenecks in multi-core cluster and explore approaches to alleviate or eliminate the bottlenecks. The hope is to provide insights to application writers on how to optimize algorithms and/or data distribution for multicore clusters. Chai e.al. have also designed an example to demonstrate the effect of multi-core aware algorithm.
- **Scalability Tests:** A set of experiments carried out to study the scalability of multi-core clusters.

The focus has been to reduce the polling overhead and include information regarding core topology for better data locality. Improvements to memory usage in the multi-core system will clearly aid any additional efforts to optimize the system-wide performance, including our own. In particular, previous research has focused on optimizing memory access and using overlapping communications between nodes and internally to improve performance which has some similarities to proposed multi-core optimizations [24, 25]. Zhang et al. proposed a similar system for mapping processes using topology information for SMP clusters as well [26]. The most relevant work from the SMP group was done by Tipparaju et al. who used a nested binomial tree to limit inter-node communications on SMP clusters to one process per node and then spread information to processes [27]. Tipparaju's work is of critical importance to this research and the modified Fastest Edge First (*FEF*) heuristic can

be directly linked back to their work.

UTK has recently presented HierKNEM a kernel-assisted topology-aware collective framework which orchestrates the collaboration between multiple layers of collective algorithms [28]. The resulting scheme enables perfect overlap of intra- and inter-node communications. By considering three of the most used collective operations (Broadcast, Allgather and Reduction), they have experimentally shown that 1) the approach is immune to modifications of the underlying process-core binding; 2) it outperforms state-of-art MPI libraries (Open MPI, MPICH2 and MVAPICH2) demonstrating up to a 30x speedup for synthetic benchmarks, and up to a 3x acceleration for a parallel graph application (ASP); 3) it demonstrates a linear speedup with the increase of the number of cores per node, a paramount requirement for scalability on future many-core hardware. They accomplished this through the benefit from the overlap in the inter-node communication and careful consideration of the copy-in/copy-out approach.

2.2.2 Optimizing communication on System level

Innovative Computing Laboratory at the University of Tennessee is working on improving the method of point-to-point communications in multicore systems such that the process is structured to take advantage of shared memory structures between cores [29]. Their work is founded in using the underlying hardware topology to define a parameter set for optimization tuning at runtime. Their inclusion of topology awareness is intended for the 'message passing middleware' so that the system can be optimized without altering the standard programming model. Such improvements in point-to-point communication complement our own work on multidimensional communications.

At UIUC, there is work being done by Bhatele et al. to optimize the algorithms used in the automatic mapping for system-level communication. Through this method they intend to minimize the contention (collisions due to sharing) across the grid and maximize the amount of resources that can be shared by the MPIs. They further extend this mapping to the processes and nodes so that those nodes who communicate amongst each other are in

closer proximity. Ultimately, they work to balance the overall load of the communications being performed so that the processes are spread across the entire communication grid, as evenly as possible. Each smaller group will communicate amongst itself before moving on to another portion of the system in an effort to avoid bottle-necking the communication processes [30, 31].

2.2.3 Optimizing the MPI methods

There are other groups who have focused their research on collective communication algorithms designed with topology awareness. Their methods include two significant advantages for performance improvement [32, 33]. Firstly, the shared memory leveraged towards exchanging messages within nodes rather than point-to-point calls between nodes, secondly, that communication stages within the nodes can take advantage of not moving data across the network and therefore minimize contention [34–37]. Work is also being conducted to produce tools that will automatically discover the physical network topology and enhance the manageability of modern IP networks [38–41]. Both the benefit of shared memory within the nodes and limiting the movement of data across the network help to optimize our application with its multidimensional collective communication.

Some of the most directly relevant research discusses how a new communication model is needed for multi-core clusters [42]. Tu et al. continues this discussion showing how a more formal communication model can be developed to replace *logP* and others. They go farther to explain how multi-core clusters differ substantially from SMP clusters and why it is important to optimize multi-core clusters independently [43, 44]. Both Tu’s group and Mamidala et al. discuss how architectural designs can have dramatic impacts on the communication speeds of various collective communications [10, 45]. Nishtala and Yelick discuss an alternative approach to optimizing collective communications on multi-core clusters by automated tuning based on loosening synchronization requirements [46].

2.2.4 Multidimensional Communication & MPI

Much of the current work and research regarding the field of MPI is being conducted to improve the message passing on the two major implementations (MPICH2 [47] and Open MPI [16, 48]) of the MPI standard. The majority of this work on collective communications has been geared towards taking advantage of the new parallel architecture and using the potential for both bi- and multi-directional communications present in modern networks [49–52]. Their effort is on improving the performance of collective communication operations in MPICH for clusters connected by switched networks. For each collective operation, they implement multiple algorithms depending on the message size, with the goal of minimizing latency for short messages and minimizing bandwidth use for long messages. Thakur et al. have implemented new algorithms for all MPI collective operations. Performance results on a Myrinet-connected Linux cluster and an IBM SP indicate that, in all cases, these algorithms significantly outperform the old algorithms used in MPICH on these type clusters, and in many cases, they outperform the algorithms used in IBM’s MPI on the SP [51, 53].

This is a very useful improvement since any optimizations done to the MPI implementations, themselves will have an improvement on our work and our contributions. Even if this was not an attempt to improve the MPI directly, these algorithms would still be critical tools for any application-based solution.

2.2.5 Optimizing multidimensional neighbor communication

Some of our work will partly be an extension of the work done by Chavarria-Miranda, et al. on tile mapping for multi-dimensional neighbor communication [54]. They propose a technique to optimize the performance of applications using distributed dense arrays and characterized by a nearest-neighbor communication profile by exploiting the topology of SMP clusters. The topological information is used to map array tiles to processors to reduce network communication and improve utilization of shared memory for inter-process communication. They demonstrate the potential benefits of using the SMP-aware mapping through

a simulation, as well as a real application solving a wind-driven ocean circulation model on an IBM SP (Scalable POWERparallel).

While they worked to improve point-to-point multidimensional communication, we take this concept even further and extend it to that of collective communications. Clearly, their initial steps which improved the multidimensional neighbor communications are beneficial for our own efforts.

CHAPTER 3

THE EXTENDED PARALLEL PING-PONG TEST

In this chapter we present an innovated approach to an extended ping-pong test with the purpose of determining the raw characteristics of the network. These extended tests provide information about the network characteristics both internally between pairs of cores on the same node, and externally between pairs of cores on different nodes. These tests provide vital information for the application-level redistribution of the process-to-core ordering, results that will be used in later chapters.

3.1 Performance Measuring and Benchmarking

An early and often important task is to measure, or benchmark, the performance of the communication network with either a so called ping-pong communication test or a benchmark suite [55, 56]. The program measures the time to send messages of different sizes between two nodes and the program consists of two parallel processes. One process sends messages to the other process, which receive the messages and immediately sends them back. The first process measures how much time the message transfer takes, using the MPI timer functions. Using the Hockney model:

$$T_{i,j}(m) = \alpha_{i,j} + \beta_{i,j}m \tag{3.1}$$

where $T_{i,j}(m)$ is the time cost for process i to communicate to process j with a message of m bytes, α is the initiation cost (latency) and $1/\beta$ describes the bandwidth. We are able to use the ping-pong test as a good way of determining the raw characteristics of the network. The network characteristics often show that for small messages the latency dominates, while for large messages the bandwidth is the limiting factor. Table 3.1 shows the ping-pong results for the local small high performance computing cluster (*RA*), and Table 3.2 shows the ping-pong results for the local lab (*Alamode*).

Table 3.1: Network Characteristics *RA-cluster* (InfiniBand Cisco 7024 IB server Switch).

Message Size	Inter-communication		Intra-communication	
	RTT	Throughput	RTT	Throughput
1 kB	17.31 μs	118.319 MB/s	10.78 μs	190.043 MB/s
2 kB	22.70 μs	180.461 MB/s	8.82 μs	464.321 MB/s
4 kB	27.23 μs	300.873 MB/s	18.12 μs	452.102 MB/s
8 kB	52.38 μs	312.788 MB/s	22.51 μs	727.961 MB/s
16 kB	77.99 μs	420.174 MB/s	41.13 μs	796.748 MB/s
32 kB	89.67 μs	730.864 MB/s	74.43 μs	880.455 MB/s
64 kB	0.14 ms	949.820 MB/s	0.13 ms	1035.517 MB/s
128 kB	0.24 ms	1092.737 MB/s	0.23 ms	1133.283 MB/s
256 kB	0.46 ms	1136.622 MB/s	0.45 ms	1159.578 MB/s
512 kB	0.86 ms	1214.829 MB/s	0.87 ms	1203.724 MB/s
1 MB	1.72 ms	1222.104 MB/s	1.69 ms	1237.335 MB/s
2 MB	3.52 ms	1190.149 MB/s	3.35 ms	1251.872 MB/s
4 MB	7.86 ms	1067.380 MB/s	6.65 ms	1261.713 MB/s
8 MB	16.41 ms	1022.415 MB/s	13.23 ms	1268.234 MB/s
16 MB	32.61 ms	1028.836 MB/s	26.44 ms	1269.202 MB/s

Table 3.2: Network Characteristics *Alamode-lab* (Fast Ethernet (1Gbps) switched network links).

Message Size	Inter-communication		Intra-communication	
	RTT	Throughput	RTT	Throughput
1 kB	56.92 μs	34.326 MB/s	4.86 μs	401.878 MB/s
2 kB	75.88 μs	51.466 MB/s	7.62 μs	512.631 MB/s
4 kB	.11 ms	69.817 MB/s	12.04 μs	651.042 MB/s
8 kB	.19 ms	84.688 MB/s	20.87 μs	747.608 MB/s
16 kB	.33 ms	94.468 MB/s	38.42 μs	813.802 MB/s
32 kB	.62 ms	100.823 MB/s	82.67 μs	755.744 MB/s
64 kB	1.39 ms	89.612 MB/s	0.13 ms	984.252 MB/s
128 kB	2.35 ms	106.537 MB/s	0.23 ms	1086.957 MB/s
256 kB	4.57 ms	109.299 MB/s	0.45 ms	1106.195 MB/s
512 kB	9.09 ms	109.964 MB/s	0.98 ms	1023.541 MB/s
1 MB	18.04 ms	111.334 MB/s	1.73 ms	1156.069 MB/s
2 MB	35.76 ms	111.659 MB/s	3.74 ms	1069.519 MB/s
4 MB	71.62 ms	111.804 MB/s	8.10 ms	987.654 MB/s
8 MB	143.2 ms	111.907 MB/s	16.03 ms	1002.506 MB/s
16 MB	281.8 ms	111.960 MB/s	31.93 ms	1004.394 MB/s

The time in the table is the *Round Trip Time* (RTT); this is the total time including the latency (time delay in the system) and the transfer time for both sending and receiving the message. The table also presents the *throughput* and not *bandwidth*. Neither system shows any doubling of RT as m doubles for small messages, an indication that latency dominates the transfer times for these messages. To find values for the latency (α) and the bandwidth ($1/\beta$), the *Least Square Method* (LSM) is first implemented to determine α :

$$\begin{aligned} slope &= \frac{\sum_{i=1}^{\|m\|} (m - \bar{m}) \cdot (RTT - \overline{RTT})}{\sum_{i=1}^{\|m\|} (m - \bar{m})^2} \\ 2 \cdot \alpha &= \overline{RTT} - slope \cdot \bar{m} \end{aligned} \quad (3.2)$$

$\|m\|$ being the number of different message sizes, \bar{m} the average *size* of a message and \overline{RTT} the average round-trip time. We only want to use message sizes where the latency dominates the transfer time ($m = \{1, 2, 4, 8, 16, 32, 64 \text{ kB}\}$).

The next step is to determine the bandwidth ($1/\beta$). There are two common methods: either let the bandwidth be the slope of the regression line for the larger message sizes ($m = \{128 \text{ kB}, 256 \text{ kB}, 512 \text{ kB}, 1 \text{ MB}, 2 \text{ MB}, 4 \text{ MB}, 8 \text{ MB}, 16 \text{ MB}\}$):

$$1/\beta = 2 \cdot \frac{\sum_{i=1}^{\|m\|} (m - \bar{m}) \cdot (RTT - \overline{RTT})}{\sum_{i=1}^{\|m\|} (m - \bar{m})^2} \quad (3.3)$$

or use a message with a size that is less than or equal to the *TCP* window size (usually 16 MB). A fixed number of messages are then sent, the receiver waits until it has received all the messages and then sends a response. The bandwidth can then be estimated to:

$$1/\beta = \frac{m}{\overline{RTT}/\text{number of messages} - \alpha} \quad (3.4)$$

Both methods are valid and give an accurate value of the bandwidth. The second method helps determine the maximum sustained data rate that can be achieved at the network level, and is therefore preferred.

One of the drawbacks on the standard test is that it does not provide any information regarding the network characteristics when communication between multiple pairs occurs

simultaneously. The test is normally done by picking a single core from each node and do latency, bandwidth and bidirectional bandwidth tests between these cores. Sometimes a more exhaustive test is done by pairing cores on the different nodes and test them sequentially. We suggest to extend the standard ping-pong test into two steps: first, to test both the internal and external network characteristics by testing both between pairs of cores on the same node and between pairs on different nodes. Second, introducing two new functions that test the latency and the bandwidth between multiple pairs in parallel. The following list is a suggested outline of a more extensive benchmarking of the network characteristics, that in our opinion is also more accurate:

- Latency Test
- Bandwidth Test
- Bidirectional Bandwidth Test
- Latency Test Multiple Pairs
- Bandwidth Test Multiple Pairs

3.2 Latency Test

The latency tests (Algorithm 3.1) are carried out using the blocking versions of the MPI functions (`MPI_Send()` and `MPI_Recv()`). The sender sends a message with a given data size to the receiver and waits for a reply. The receiver receives the message from the sender and sends back a reply with the same data size. The reply message could be either a carbon copy of the received message or predetermined message of a given size to allow it to check for transmission errors. This ping-pong test is sequentially repeated over all pairs with each pair performing many iterations of the test (see Table 3.3). Average one-way latency numbers are obtained for both internal and external communication using Equation 3.2.

Algorithm 3.1: Parallel Ping-Pong: Latency Test

Input: A *Sender* and *Receiver* pair

Output: A vector of transfer times

```
1  $size_{\min} \leftarrow$  minimum message size
2  $size_{\max} \leftarrow$  maximum message size
3 for  $i \leftarrow 0$  to number of iterations do
4   for  $j \leftarrow size_{\min}$  to  $size_{\max}$  do
5     Create a SEND buffer of  $size = j$ 
6     Create a RECEIVE buffer of  $size = j$ 
7     MPI_Barrier(MPI_COMMUNICATOR)
8      $start \leftarrow$  Wall Time
9     if Sender then
10      MPI_Send(SEND, ..., MPI_COMMUNICATOR)
11      MPI_Recv(RECEIVE, ..., MPI_COMMUNICATOR)
12    end
13    else
14      MPI_Recv(RECEIVE, ..., MPI_COMMUNICATOR)
15      MPI_Send(SEND, ..., MPI_COMMUNICATOR)
16    end
17     $stop \leftarrow$  Wall Time
18     $time \leftarrow time + (stop - start)$ 
19  end
20 end
21 MPI_Barrier(MPI_COMMUNICATOR)
22 Display the vector of transfer times
```

Table 3.3: Example of a Round-Trip Time Matrix *Alamode-lab*.

0.000	0.021	0.021	0.026	0.328	0.315	0.291	0.284
0.030	0.000	0.032	0.029	0.305	0.287	0.291	0.317
0.022	0.038	0.000	0.034	0.338	0.292	0.287	0.301
0.028	0.029	0.030	0.000	0.339	0.303	0.282	0.296
0.265	0.224	0.265	0.254	0.000	0.032	0.033	0.032
0.258	0.222	0.264	0.224	0.041	0.000	0.031	0.029
0.263	0.225	0.265	0.224	0.030	0.034	0.000	0.027
0.263	0.225	0.265	0.225	0.038	0.036	0.026	0.000

Message size 16 kB, time (*ms*)

3.3 Bandwidth Tests

The objective of the bandwidth tests is to determine the maximum sustained data rate that can be achieved at the network level. The tests use non-blocking receive version (`MPI_Irecv()`), but the standard send version (`MPI_Send()`) of MPI functions. The sender sends a fixed number of back to back messages to the receiver and then waits for a reply. These messages are less than or equal to window size. The receiver waits to receive all the messages before sending a reply. This process is repeated for several iterations and the bandwidth is calculated based on the elapsed time (from the time sender sends the first message until the time it receives the reply back from the receiver) and the number of bytes sent by the sender. The test is once again sequentially repeated over all pairs with each pair performing many iterations of the test (Algorithm 3.2). Average bandwidth values are obtained for both internal and external communications using either Equation 3.3 or Equation 3.4.

3.4 Bidirectional Bandwidth Test

This test is similar to the bandwidth test, with the objective to measure the maximum sustainable aggregate bandwidth by two groups. Both the cores of the pair involved send out a fixed number of back-to-back messages and wait for the reply (Algorithm 3.3). The test is sequentially repeated over all pairs and with several iterations per pair.

3.5 Latency Test Multiple Pairs

This test is very similar to the single pair latency test, however, at the same instant multiple pairs are performing the same test simultaneously. The test is done in two different configurations: 1) External test: this tests the communication between cores on different nodes, and 2) Internal test: tests the communication between cores on the same node (Figure 3.1).

The number of pairs are increased incrementally, beginning with a single pair and ending with all cores involved in the test sending and receiving in parallel.

Algorithm 3.2: Parallel Ping-Pong: Bandwidth Test

Input: A *Sender* and *Receiver* pair

Output: A vector of transfer times

```
1 size ← fixed size less than window size
2 for i ← to number of iterations do
3   Create a SEND buffer of size
4   Create a RECEIVE buffer of size
5   MPI_Barrier(MPI_COMMUNICATOR)
6   MPI_Irecv(RECEIVE, ..., MPI_COMMUNICATOR, REQUEST)
7   if Sender then
8     start ← Wall Time
9     for j ← 0 to number of messages do
10    | MPI_Send(SEND, ..., MPI_COMMUNICATOR)
11    end
12    MPI_Wait(REQUEST, STATUS)
13    stop ← Wall Time
14    time ← time + (stop − start)
15  end
16  else
17    for j ← 0 to number of messages - 1 do
18    | MPI_Wait(REQUEST, STATUS)
19    | MPI_Irecv(RECEIVE, ..., MPI_COMMUNICATOR, REQUEST)
20    end
21    MPI_Send(SEND, ..., MPI_COMMUNICATOR)
22  end
23 end
24 MPI_Barrier(MPI_COMMUNICATOR)
25 Display the vector of transfer times
```

Algorithm 3.3: Parallel Ping-Pong: Bidirectional Bandwidth Test

Input: A *Sender* and *Receiver* pair
Output: A vector of transfer times

- 1 $size \leftarrow$ fixed size less than window size
- 2 **for** $i \leftarrow 0$ to number of iterations **do**
- 3 Create a SEND buffer of $size$
- 4 Create a RECEIVE₁ buffer of $size$
- 5 Create a RECEIVE₂ buffer of $size$
- 6 MPI_Irecv(RECEIVE₂, ..., MPI_COMMUNICATOR, REQUEST₁)
- 7 MPI_Barrier(MPI_COMMUNICATOR)
- 8 $start \leftarrow$ Wall Time
- 9 **for** $j \leftarrow 0$ to number of messages **do**
- 10 MPI_Irecv(RECEIVE₁, ..., MPI_COMMUNICATOR, REQUEST₂)
- 11 MPI_Send(SEND, ..., MPI_COMMUNICATOR)
- 12 MPI_Wait(REQUEST₂, STATUS)
- 13 **end**
- 14 MPI_Send(SEND, ..., MPI_COMMUNICATOR)
- 15 MPI_Wait(REQUEST₁, STATUS)
- 16 $stop \leftarrow$ Wall Time
- 17 $time \leftarrow time + (stop - start)$
- 18 **end**
- 19 MPI_Barrier(MPI_COMMUNICATOR)
- 20 Display the vector of transfer times

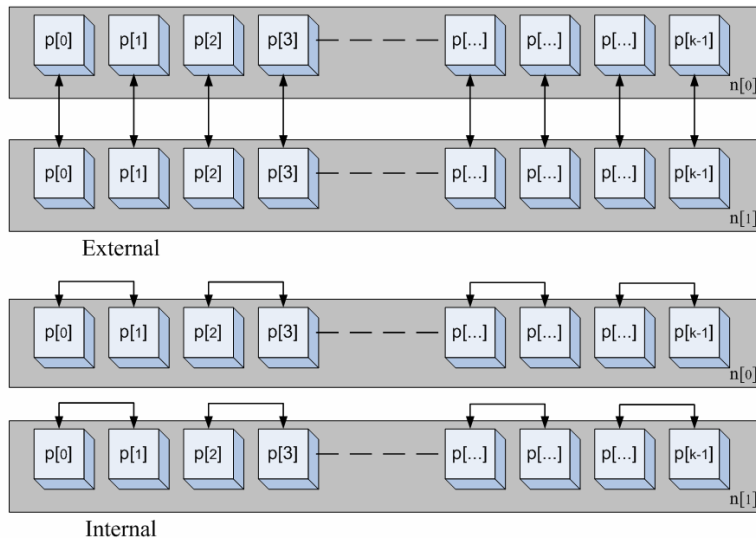


Figure 3.1: Multiple Pairs Parallel Ping-Pong Test with two different nodes each containing k cores.

Algorithm 3.4: Parallel Ping-Pong: Latency Test Multiple Pairs

Input: An array of *Sender* and an array of *Receiver*

Output: A vector of transfer times

```
1  $size_{\min} \leftarrow$  minimum message size
2  $size_{\max} \leftarrow$  maximum message size
3 for  $i \leftarrow 0$  to number of elements in Sender do
4   for  $j \leftarrow 0$  to number of iterations do
5     for  $k \leftarrow size_{\min}$  to  $size_{\max}$  do
6       /* Is the core the  $i^{th}$  or lower element in either the Send or
7         Receive array? */
8       if  $Sender_{rank} \leq i$  or  $Receiver_{rank} \leq i$  then
9         Create a SEND buffer of  $size = k$ 
10        Create a RECEIVE buffer of  $size = k$ 
11        MPI_Barrier(MPI_COMMUNICATOR)
12         $start \leftarrow$  Wall Time
13        if Sender then
14          MPI_Send(SEND, ..., MPI_COMMUNICATOR)
15          MPI_Recv(RECEIVE, ..., MPI_COMMUNICATOR)
16        end
17        else
18          MPI_Recv(RECEIVE, ..., MPI_COMMUNICATOR)
19          MPI_Send(SEND, ..., MPI_COMMUNICATOR)
20        end
21         $stop \leftarrow$  Wall Time
22         $time \leftarrow time + (stop - start)$ 
23      end
24    end
25  end
26  MPI_Barrier(MPI_COMMUNICATOR)
27  Display the vector of transfer times
```

3.6 Bandwidth Test Multiple Pairs

The multi-pair bandwidth and message rate test evaluates the aggregate uni-directional bandwidth and message rate between multiple pairs of processes. Each of the sending processes sends a fixed number of messages (the window size) back-to-back to the paired receiving process before waiting for a reply from the receiver. This process is repeated for several iterations. The objective of this benchmark is to determine the achieved bandwidth and message rate from one node to another node with a configurable number of processes running on each node.

3.7 Evaluation

To get results that would reflect as many possible applications as possible, we used four different different hardware configurations:

3.7.1 System Specifications

- *Alamode Lab* a local lab cluster of 23 machines with 2x Dual Core Opteron 2218 2.6GHz, 5.2 GFlops, 8 GB and `Open MPI`. Fast Ethernet (1Gbps) switched network links.
- *RA* a local small high performance computing cluster with Dell Intel quad-core, dual socket (Clovertown E5355 2.76 GHz, 10.6 GFlops), 16 GB and `OpenMPI`. Cisco 7024 IB server Switch.
- *Jaguar* a Cray XT5-HE six-core, dual socket system (AMD 2.6 GHz Istanbul-6, 10.4 GFlops), 16 GB and `MPICH2 1.0.6 (MPT 3.1.02)`. Cray SeaStar2+ router.
- *BlueDrop* a single node system with an IBM eight-core, quad socket system (Power7 3.5 GHz, 256 GFlops) with 128 GB memory.

Not all systems were used in all the experiments, but the Extended Parallel Ping-Pong Test was done on all systems. All results except for the BlueDrop are presented here.

Algorithm 3.5: Parallel Ping-Pong: Bandwidth Test Multiple Pairs

Input: A *Sender* and *Receiver* pair
Output: A vector of transfer times

```
1 size ← fixed size less than window size
2 for i ← 0 to number of elements in Sender do
3   for j ← 0 to number of iterations do
4     /* Is the core the ith or lower element in either the Send or
5      Receive array? */
6     if Senderrank ≤ i or Receiverrank ≤ i then
7       Create a SEND buffer of size
8       Create a RECEIVE buffer of size
9       MPI_Barrier(MPI_COMMUNICATOR)
10      MPI_Irecv(RECEIVE, ..., MPI_COMMUNICATOR, REQUEST)
11      start ← Wall Time
12      if Sender then
13        start ← Wall Time
14        for k ← 0 to number of messages do
15          MPI_Send(SEND, ..., MPI_COMMUNICATOR)
16        end
17        MPI_Wait(REQUEST, STATUS)
18        stop ← Wall Time
19        time ← time + (stop − start)
20      end
21      else
22        for k ← 0 to number of messages - 1 do
23          MPI_Wait(REQUEST, STATUS)
24          MPI_Irecv(RECEIVE, ..., MPI_COMMUNICATOR, REQUEST)
25        end
26        MPI_Send(SEND, ..., MPI_COMMUNICATOR)
27      end
28    end
29  end
30 MPI_Barrier(MPI_COMMUNICATOR)
31 Display the vector of transfer times
```

BlueDrop is a single node system and the system characteristics will only be presented in the multiple pair section.

3.7.2 Latency and Bandwidth Tests

The Latency-test on the *Alamode* slightly lower values than expected for a Fast Ethernet system; $\alpha_E = 8.57 \mu\text{sec}$ and $\alpha_I = 9.68 \mu\text{sec}$ (Figure 3.2(a)). The bandwidth for both internal and external communication was within the expected range; $\beta_E = 118.93 \text{ MB/sec}$ and $\beta_I = 1046.3 \text{ MB/sec}$ (see Figure 3.2(b)).

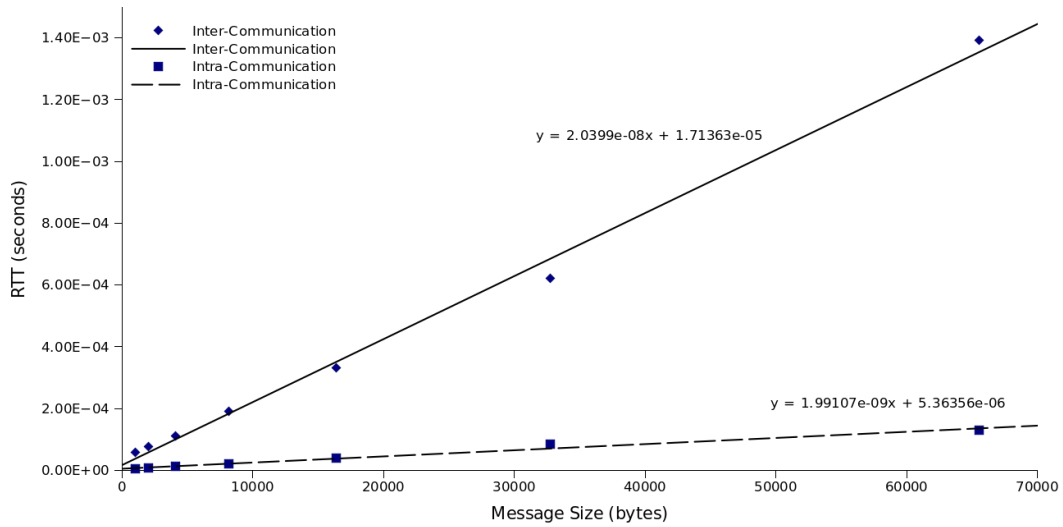
The external Latency on *RA* was slightly higher than expected $\alpha_E = 13.52 \mu\text{sec}$, which is about 10 times the expected value. It could be explained by other network activities, but it could also have been a bad link (see Figure 3.3(a)). The bandwidth had the expected results for a $4x$ IB switch (see Figure 3.3(b)).

The Latency values for *Jaguar* were within the expected range; $\alpha_E = 1.33 \mu\text{sec}$ and $\alpha_I = .370 \mu\text{sec}$ (see Figure 3.4(a)). The bandwidth is the highest of all three systems (see Figure 3.4(b)).

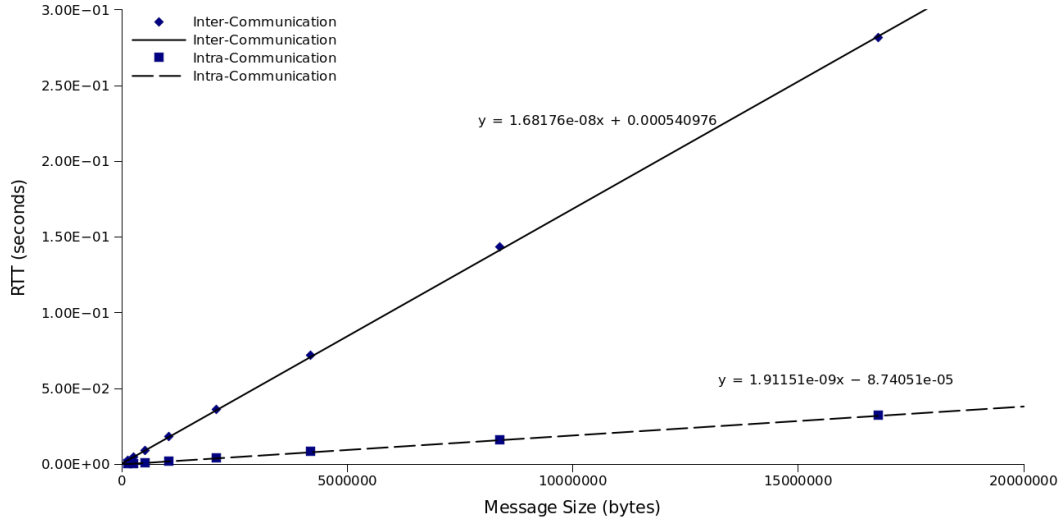
The results for the linear regressions on all three systems are presented in Table 3.4. All three systems had double data rate, meaning that the bandwidth doubled during the *Bidirectional*-tests.

Table 3.4: The combined results from the Linear Regressions.

System	<i>Inter-Communication</i>	<i>Intra-Communication</i>
Alamode	$\alpha_E = 8.57 \cdot 10^{-6} \text{ s}$	$\alpha_I = 9.68 \cdot 10^{-6} \text{ s}$
	$1/\beta_E = 118.93 \text{ MB/sec}$	$1/\beta_I = 1046.3 \text{ MB/sec}$
RA	$\alpha_E = 13.52 \cdot 10^{-6} \text{ s}$	$\alpha_I = 4.33 \cdot 10^{-6} \text{ s}$
	$1/\beta_E = 1021.3 \text{ MB/sec}$	$1/\beta_I = 1271.0 \text{ MB/sec}$
Jaguar	$\alpha_E = 1.33 \cdot 10^{-6} \text{ s}$	$\alpha_I = 3.70 \cdot 10^{-7} \text{ s}$
	$1/\beta_E = 1627.5 \text{ MB/sec}$	$1/\beta_I = 2218.8 \text{ MB/sec}$

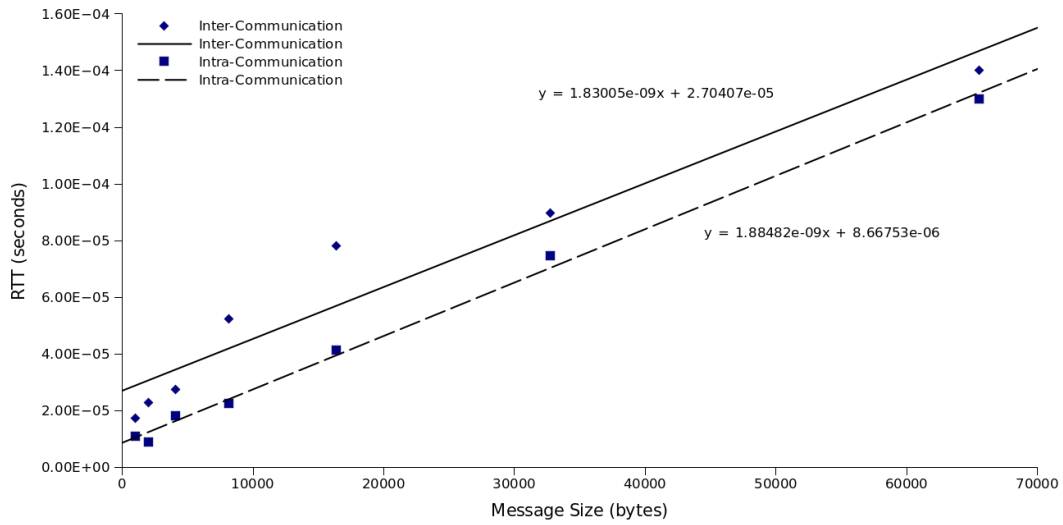


(a) Latency (α) Tests ($m = \{1, 2, 4, 8, 16, 32, 64 \text{ kB}\}$)

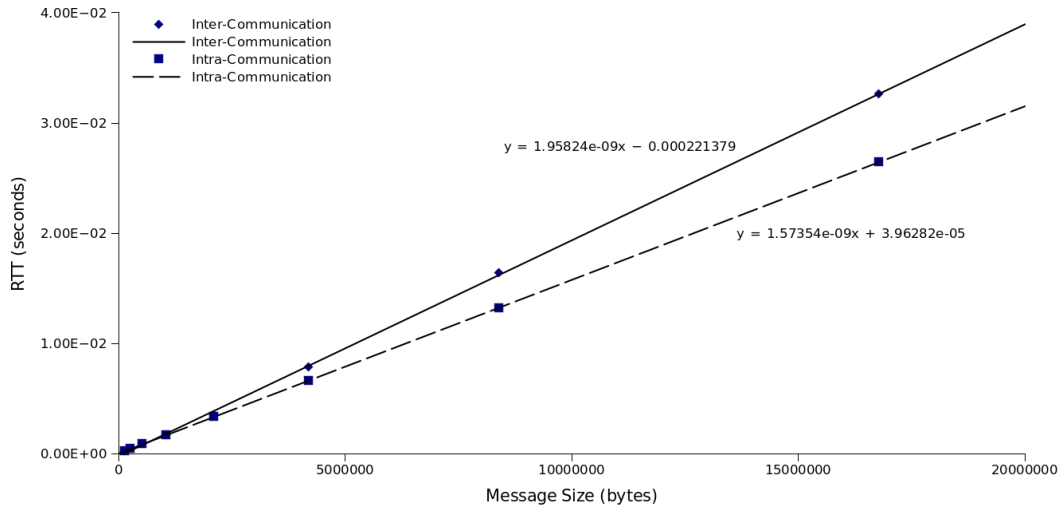


(b) Bandwidths ($1/\beta$) Tests ($m = \{128 \text{ kB}, 256 \text{ kB}, 512 \text{ kB}, 1 \text{ MB}, 2 \text{ MB}, 4 \text{ MB}, 8 \text{ MB}, 16 \text{ MB}\}$)

Figure 3.2: Linear Regression *Alamode*, both *Inter-* and *Intra-*Communication.

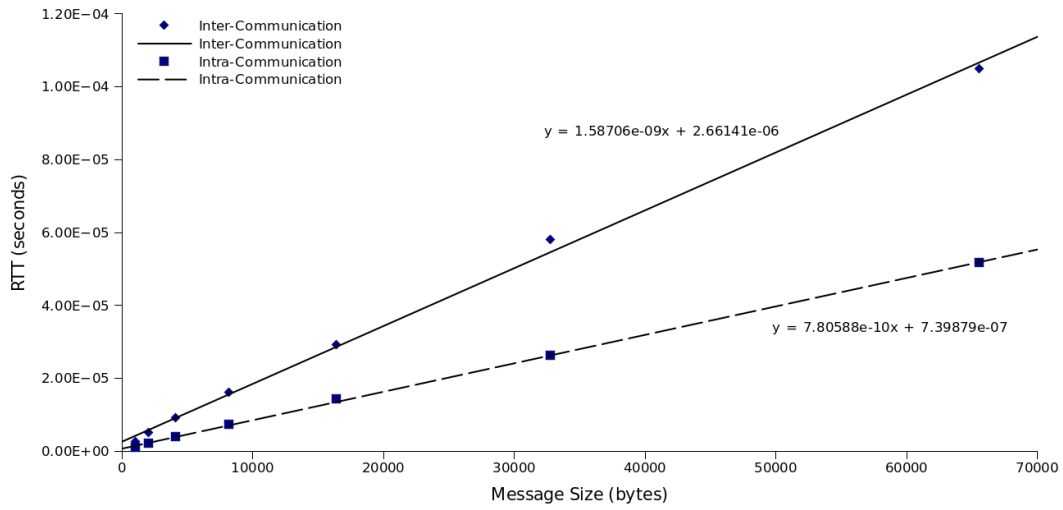


(a) Latency (α) Tests ($m = \{1, 2, 4, 8, 16, 32, 64 \text{ kB}\}$)

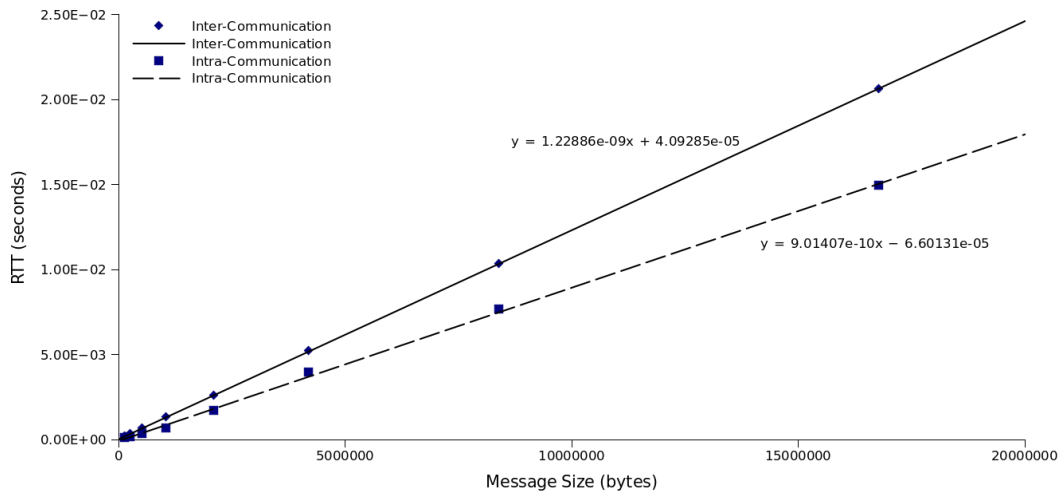


(b) Bandwidths ($1/\beta$) Tests ($m = \{128 \text{ kB}, 256 \text{ kB}, 512 \text{ kB}, 1 \text{ MB}, 2 \text{ MB}, 4 \text{ MB}, 8 \text{ MB}, 16 \text{ MB}\}$)

Figure 3.3: Linear Regression RA , both *Inter*- and *Intra*-Communication.



(a) Latency (α) Tests ($m = \{1, 2, 4, 8, 16, 32, 64 \text{ kB}\}$)



(b) Bandwidths ($1/\beta$) Tests ($m = \{128 \text{ kB}, 256 \text{ kB}, 512 \text{ kB}, 1 \text{ MB}, 2 \text{ MB}, 4 \text{ MB}, 8 \text{ MB}, 16 \text{ MB}\}$)

Figure 3.4: Linear Regression *Jaguar*, both *Inter-* and *Intra-*Communication.

3.7.3 Latency and Bandwidth Tests Multiple Pairs

The results from the *Round-Trip Time* tests show an expected increase in *RTT* as the amount of participating pairs increases (see Table 3.5, Table 3.6 and Table 3.7).

Table 3.5: Latency and Bandwidth Tests Multiple Pairs on *Alamode* $m = 8$ MB

Round-Trip Time (<i>msec</i>)				
Cores	External	Internal	Cores	External
1 ↔ 1	136.9	13.702	3 ↔ 3	257.1
2 ↔ 2	203.3	19.074	4 ↔ 4	334.2

Table 3.6: Latency and Bandwidth Tests Multiple Pairs on *RA* $m = 8$ MB

Round-Trip Time (<i>msec</i>)				
Cores	External	Internal	Cores	External
1 ↔ 1	13.024	12.536	5 ↔ 5	21.842
2 ↔ 2	14.091	22.324	6 ↔ 6	24.909
3 ↔ 3	16.515	29.360	7 ↔ 7	31.290
4 ↔ 4	16.236	41.863	8 ↔ 8	38.273

Table 3.7: Latency and Bandwidth Tests Multiple Pairs on *Jaguar* $m = 8$ MB

Round-Trip Time (<i>msec</i>)				
Cores	External	Internal	Cores	External
1 ↔ 1	9.7116	7.5703	7 ↔ 7	41.201
2 ↔ 2	10.475	12.853	8 ↔ 8	41.650
3 ↔ 3	17.780	13.816	9 ↔ 9	43.057
4 ↔ 4	24.346	14.883	10 ↔ 10	46.910
5 ↔ 5	32.112	18.028	11 ↔ 11	48.103
6 ↔ 6	34.533	17.098	12 ↔ 12	55.741

It is more interesting to look at the results from the tests in terms of the total bandwidth $(p \cdot m)/RTT$. Looking at the *Alamode*-lab (see Table 3.8) we can see that the *External Total Bandwidth* is slightly increasing as we increase the number of pairs from one to four (increasing p from two to eight). *Internal Total Bandwidth* also seems to stay fairly constant.

Table 3.8: Total Bandwidth Multiple Pairs on *Alamode* $m = 8$ MB

Bandwidth (MB/sec)				
Cores	External	Internal	Cores	External
1 ↔ 1	122.53	1224.4	3 ↔ 3	165.05
2 ↔ 2	195.73	1759.2	4 ↔ 4	200.82

For the *RA*-cluster we see a sharp increase in the *External Total Bandwidth* when we increase the number of pairs from one to three, and it stays fairly constant after that (see Table 3.9). *Internal Total Bandwidth* stay fairly constant. Our interpretation is that the *RA*-cluster has three external channels for communication and a single internal channel.

Table 3.9: Total Bandwidth Multiple Pairs on *RA* $m = 8$ MB

Bandwidth (MB/sec)				
Cores	External	Internal	Cores	External
1 ↔ 1	1288.1	1338.3	5 ↔ 5	3840.5
2 ↔ 2	2381.3	1503.1	6 ↔ 6	4041.3
3 ↔ 3	3047.6	1714.3	7 ↔ 7	3753.3
4 ↔ 4	4133.4	1603.1	8 ↔ 8	3506.9

The *Jaguar*-machine has a sharp increase in the *External Total Bandwidth* when we increase the number of pairs from one to two, and it stays fairly constant after that (see Table 3.10). *Internal Total Bandwidth* stay has a constant bandwidth increase as we increase the number of pairs from one to six (p from two to twelve). Our interpretation is that the *Jaguar*-cluster has three external channels for communication and the internal bandwidth follows a linear regression $1/\beta_I = 1385 + 724 \cdot p/2$ MB/s.

The Latency tests showed that all systems have an increase of the latency times in multi pair compared the single pair, the increase is not on the magnitude of any factor, but it is still significant. (see Table 3.11). The total bandwidth in cases of a single channel it was about about $1.5 \cdot$ single pair bandwidth, and for multiple channels was close to $c \cdot$ single pair bandwidth.

Table 3.10: Total Bandwidth Multiple Pairs on *Jaguar* $m = 8$ MB

Bandwidth (MB/sec)				
Cores	External	Internal	Cores	External
1 ↔ 1	1727.5	2216.2	7 ↔ 7	2850.4
2 ↔ 2	3203.4	2610.6	8 ↔ 8	3222.5
3 ↔ 3	2830.8	3642.9	9 ↔ 9	3506.9
4 ↔ 4	2756.5	4509.2	10 ↔ 10	3576.5
5 ↔ 5	2612.3	4653.0	11 ↔ 11	3836.6
6 ↔ 6	2914.9	5887.4	12 ↔ 12	3611.8

Table 3.11: The combined results from the Multiple Pair test.

System	<i>Inter-Communication</i>	<i>Intra-Communication</i>
Alamode	$\alpha_E = 9.76 \cdot 10^{-6} s$	$\alpha_I = 10.22 \cdot 10^{-6} s$
Total Bandwidth	$1/\beta_E = 171.0$ MB/sec	$1/\beta_I = 1491.8$ MB/sec
RA	$\alpha_E = 9.41 \cdot 10^{-6} s$	$\alpha_I = 9.53 \cdot 10^{-6} s$
Total Bandwidth	$1/\beta_E = 2712.6$ MB/sec	$1/\beta_I = 1529.7$ MB/sec
Jaguar	$\alpha_E = 2.04 \cdot 10^{-6} s$	$\alpha_I = 4.05 \cdot 10^{-7} s$
Total Bandwidth	$1/\beta_E = 2629.5$ MB/sec	$1/\beta_I = 1385 + 724 \cdot p/2$ MB/sec

3.7.4 The BlueDrop System

BlueDrop is not a system, it is a single *IBM Power7* node at The National Center for Supercomputing Applications. It was supposed to be the hardware foundation for the *Blue Waters* project, until they changed and decided to use the *AMD Opteron 6200 Series* processor instead. The *IBM Power7* model 780 has four chips with eight cores per chip for a total of 32 cores clocked at 4.14 GHz. Further, the *Power 7* has an Instruction Sequence Unit that has the capacity of dispatching six instructions per cycle to more than one queue. Up to eight instructions per cycle can be issued to the twelve Instruction Execution units. *IBM Power7* also implements what is known as Aggressive Out-of-Order instruction execution to increase the use of all available execution paths. The theoretical performance for a chip is 265 GFLOPS (4.41 GHz · 8 instructions/cycle · 8 cores/chip).

The results from *BlueDrop* show that the 32 cores form two distinct groups of 16 cores. The difference in bandwidth between communication within a group and between two processes in different groups is statistically significant (see Table 3.12).

Table 3.12: RTT for Ping-Pong test on *BlueDrop* ($m = 512 \text{ kB}$).

rank	time (ms)						
	0	1	2	3	4	5	...
0	0.000	0.528	0.299	0.306	0.283	0.455	...
1	0.529	0.000	0.505	0.558	0.500	0.365	...
2	0.238	0.497	0.000	0.439	0.297	0.455	...
3	0.335	0.543	0.351	0.000	0.333	0.513	...
5	0.220	0.494	0.311	0.308	0.000	0.458	...
5	0.455	0.416	0.461	0.544	0.455	0.000	...
...

The other important observation is that the *RTT* stays constant at about 1.5 of the single pair time during the multi-pair tests (see Table 3.13). This means that the bandwidth for *BlueDrop* follows the same linear regression as *Jaguar* for processes within the same group.

Table 3.13: The RTT for Multi-pair bandwidth test between processes in the same group ($= 512 \text{ kB}$).

Cores	time (ms)	
	$Group_1$	$Group_2$
1 ↔ 1	0.33	0.34
2 ↔ 2	0.44	0.46
3 ↔ 3	0.48	0.42
4 ↔ 4	0.44	0.43
5 ↔ 5	0.42	0.42
6 ↔ 6	0.46	0.43
7 ↔ 7	0.44	0.47
8 ↔ 8	0.45	0.46

The RTT has a slow increase when sending messages in parallel between processes in the two groups, and there seems to be no distinct number of pairs where there is a rapid increase as in any of the other systems (see Table 3.14). Notice that it takes twice as long for any

given pair to exchange their data, but the total amount of data sent in parallel has increased by a factor of 16 while the *RTT* has only doubled.

Table 3.14: The RTT for Multi pair bandwidth test between processes in different groups (= 512 *kB*).

Cores	time (<i>ms</i>)	Cores	time (<i>ms</i>)
1 ↔ 1	0.50	9 ↔ 9	0.78
2 ↔ 2	0.54	10 ↔ 10	0.43
3 ↔ 3	0.58	11 ↔ 11	0.82
4 ↔ 4	0.56	12 ↔ 12	0.86
5 ↔ 5	0.62	13 ↔ 13	0.92
6 ↔ 6	0.66	14 ↔ 14	0.93
7 ↔ 7	0.74	15 ↔ 15	0.97
8 ↔ 8	0.78	16 ↔ 16	1.14

The combined results for the *BlueDrop* shows a latency between processes in the same group to be: $\alpha_I = 1.14 \mu s$ and the bandwidth $1/\beta_I = 3260 \text{ MB/s}$. The latency between processes in different groups $\alpha_E = 1.20 \mu s$ and the bandwidth $1/\beta_E = 1646 \text{ MB/s}$. For the multiple pair test the average latency was $\alpha = 6.71 \mu s$ and we can expect each pair to have an average bandwidth of $\beta = 2510 \text{ MB/s}$.

CHAPTER 4

SINGLE NODE OPTIMIZATION

In this chapter, we present a re-mapping algorithm for a single node. We are implementing and extending work done on non-homogeneous clusters into working algorithms for a single node using unmodified standard MPI communicators. We apply our application-level multicore-aware process-to-core re-mapping scheme at runtime and are able to achieve optimal performance for communication on binomial trees.

4.1 Communication on Shared Memory Parallel Node

When conducting extended Ping-Pong tests that contain both a One-to-One and Parallel Ping-Pong Test on many of the modern systems, both expected and surprising results are revealed. The extended test makes it possible to measure how many messages can be sent both Externally and Internally in parallel without any significant loss in bandwidth. The One-to-One extension also makes it possible to tell which cores have close connection to each other. A noticeable result is that on many of the systems the cores on a single node can be grouped into at least two distinct groups (Table 4.1), *Group 1* = {0, 2, 3, 5} and *Group 2* = {1, 4, 6, 7}. This is not surprising as a node many times consists of two physical chips on these systems (Figure 4.1).

The results on IBM Power7 (Figure 4.2) is even more pronounced, with almost a doubling in bandwidth between the groupings (Table 4.2);

$$G1 = \{0, 2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 17, 20, 21, 26, 30\}$$

and

$$G2 = \{1, 5, 7, 11, 13, 16, 18, 19, 22, 23, 24, 25, 27, 28, 29, 31\}.$$

What used to be a common assumption with a single-core solution, that the bandwidth between cores are equal and that there are sufficient amount of channels between the cores to send unlimited amount of parallel messages at the same time, is no longer true. Even

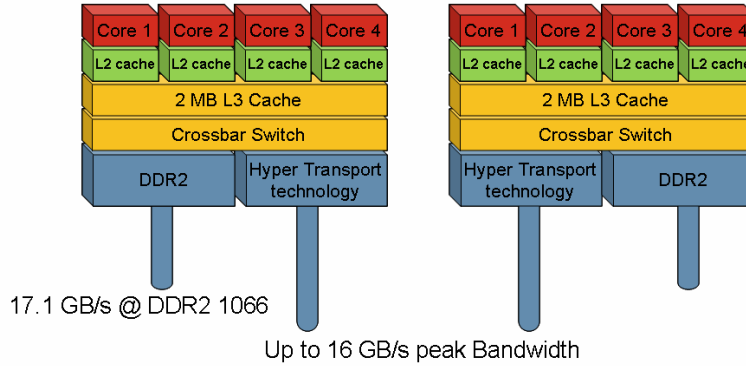


Figure 4.1: Example of a modern multicore node; two AMD Phenom X4 Quad-Core Processors.

Table 4.1: An example of a Cost Matrix (round trip travel-times) from the RA supercomputing cluster at the School of Mines using 8 cores and a scan size of 16 kB (μs).

0.00	32.74	30.69	30.62	32.42	30.63	32.25	32.79
32.37	0.00	32.97	32.90	30.35	32.56	30.59	30.41
30.60	33.50	0.00	30.67	32.68	30.58	32.52	32.81
30.34	33.37	30.75	0.00	32.73	30.46	32.78	33.01
32.67	30.56	32.81	32.62	0.00	32.91	30.53	30.50
30.61	33.24	30.97	30.67	32.87	0.00	32.76	32.83
32.70	30.51	32.65	32.86	30.62	32.85	0.00	30.49
32.91	30.63	32.87	32.72	30.50	32.97	30.67	0.00

Table 4.2: An example of a Cost Matrix (round trip travel-times) from the Power7 node using 32 cores and a scan size of 16 kB (μs).

0.000	17.88	9.42	9.89	9.57	18.48	9.64	...
17.84	0.000	18.48	18.52	17.90	9.76	18.08	...
9.91	18.95	0.000	9.53	9.78	18.42	9.43	...
9.46	17.87	9.62	0.000	9.74	17.88	9.63	...
9.91	18.32	9.56	9.51	0.000	17.93	9.66	...
18.03	9.63	18.11	17.84	18.32	0.000	18.21	...
9.67	18.06	9.71	9.43	9.58	18.12	0.000	...
...

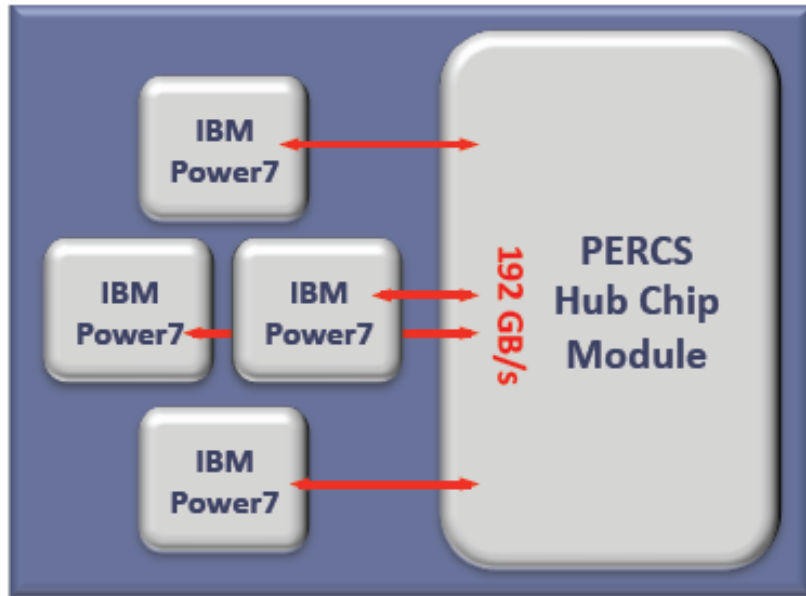


Figure 4.2: Example of a Power7 Node with four Processors.

communication between cores on the same node has different bandwidth and there are limited amounts of both internal and external channels on each node. These differences have an impact on the performance of the collective communication routines and there is room for improvements if there is some understanding about the hardware where the algorithms are implemented [57–60]. Our goal is to show an application-layer implementation that optimizes the collective communication routines that utilizes binomial tree structures for their implementations. Examples of communication routines that use this type of structure are: 1) Broadcast with small messages on small grids (small messages are defined during the installation of the MPI, *rpi_tcp_short* sets the threshold for when MPI switches between the short- and long-message protocol; most systems uses 64 kB as the threshold), and 2) the Gather and Scatter routines in the MPI-library. We focus on an MPI-implementation on the application layer because we try to avoid a hybrid solution between *openMP* and *MPI*, and optimized communication on the node-level are in many cases the foundation for optimized communication on the system level[61–63].

Bhat *et al.* shows that the heterogeneous Hockney communication model (Equation 3.1) can be used to represent point-to-point communications between processes and not only nodes [64]. In our case, there will be only one MPI process running on each available core in the node. We use the concept of MPI processes when describing our models instead of nodes, processors or cores.

The short-message protocol (binomial tree structure) advantage is that it reduces the amount of transfers per message. The number of steps to transfer a message in a p -process system is equal to:

$$\text{steps} = \lceil \log_2 p \rceil$$

We can make some general assumptions when we examine this type of communication. We know that the difference in the binary address between the sender and receiver in the first step of scatter is a one in the Most Significant Bit (MSB), and the last step a one in the Least Significant Bit (LSB) of their binary rank (see Appendix). We can also assume that in each step of the scatter the total amount of data sent will be the same (Figure 4.3 and Table 4.3).

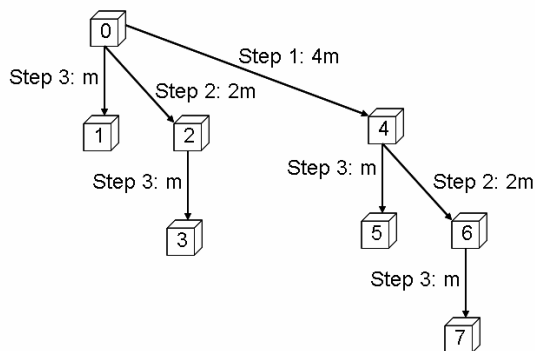


Figure 4.3: Example of a binomial tree, an array of $8 \cdot m$ is scattered to eight processes, so that each process end with a segment of size m .

The example shows how an $8 \cdot m$ is scattered in three steps to 8 processes. In step 1 ($p_{[0]} \rightarrow p_{[4]}$), all the segments ($4 \cdot m$) for higher part of the binomial tree ($p_{[4]}$ to $p_{[7]}$) are sent

Table 4.3: A message of $8m$ is dispersed to eight processes.

Step	From \rightarrow To	Message size
1	$p_{[0]} \rightarrow p_{[4]}$	$4 \cdot m$
2	$p_{[0]} \rightarrow p_{[2]}$	$2 \cdot m$
	$p_{[4]} \rightarrow p_{[6]}$	$2 \cdot m$
3	$p_{[0]} \rightarrow p_{[1]}$	m
	$p_{[2]} \rightarrow p_{[3]}$	m
	$p_{[4]} \rightarrow p_{[5]}$	m
	$p_{[6]} \rightarrow p_{[7]}$	m

from $p_{[0]}$ to $p_{[4]}$. In the second step both $p_{[0]}$ and $p_{[4]}$ sends a message of size $2 \cdot m$ to $p_{[2]}$ and $p_{[6]}$ respectively for a total of $2 \cdot 2 \cdot m$. Finally, in the last step all even processes send a message of size m to all the odd processes. The end result is that in three steps $8 \cdot m$ have been dispersed to eight process by passing $4 \cdot m$ in each step.

Algorithm 4.6: Binomial Scatter

Input: a message vector m , a MPI communicator
Output: a completion status

- 1 $levels \leftarrow \lceil \log_2 \text{size of rank} \rceil$
- 2 $myrank \leftarrow$ process position in **rank** vector
- 3 **for** $step \leftarrow 0$ to $levels - 1$ **do**
- 4 $offset \leftarrow 2^{levels-(step+1)}$
- 5 **if** $myrank$ is congruent to 0 mod $2^{levels-step}$ and
- 6 $myrank + offset$ is less than size of **rank** **then**
- 7 send message to $myrank + offset$
- 8 **end**
- 9 **else if** $myrank$ is congruent to 0 mod $offset$ **then**
- 10 receive message from $myrank - offset$
- 11 **end**
- 12 **end**
- 13 Return $Status \leftarrow Complete$

Assuming a communication model with equal bandwidth between all cores and an unlimited amount of channels, we can estimate the time it takes to scatter this message to:

$$(4 \cdot m + 2 \cdot m + m) \cdot \beta = 7 \cdot m \cdot \beta$$

(ignoring the latency times). Notice that in the last step we assume that we can send $p/2$ messages in parallel, on a Power7 with 32 cores it would mean we would have the ability to send 16 messages in parallel.

We know from the Extended Parallel Ping-Pong Tests that none of the systems have an unlimited amount of channels. The fact is that all systems had a significant bandwidth loss when sending more than two messages in parallel. If we let c be the amount of message that can be sent simultaneously in parallel we can change our model and estimate the communication time to:

$$(4 \cdot m + 2 \cdot m + 4/c \cdot m) \cdot \beta = 8 \cdot m \cdot \beta$$

This does not seem much with an increase of only $(m \cdot \beta)$, but let us examine what happen when we increase to 32 cores (see Figure 4.4).

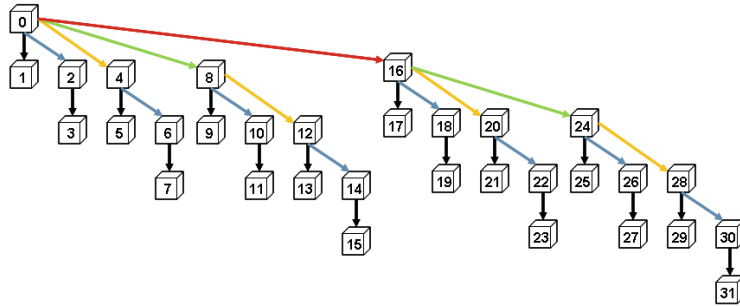


Figure 4.4: Example of a binomial tree, an array of $32 \cdot m$ is scattered to thirty two processes, so that each process end with a segment of size m .

With unlimited amount of channels:

$$(16 \cdot m + 8 \cdot m + 4 \cdot m + 2 \cdot m + m) \cdot \beta = 31 \cdot m \cdot \beta$$

With two channels:

$$(16 \cdot m + 8 \cdot m + (4 \cdot 4/c) \cdot m + (2 \cdot 8/c) \cdot m + (16/c) \cdot m) \cdot \beta = 48 \cdot m \cdot \beta$$

This means that on a 32 core node a more accurate communication model for binomial trees is almost 50% slower than the current model.

4.2 Fastest Edge First

Our first improvement is to implement the Fastest Edge First (FEF) heuristic to our multi-core node. To complete the extension of the FEF heuristic, we will modify the heterogeneous Hockney model (Equation 3.1) by replacing the β term with a direct network round trip time (RTT) measurement and assume that all cores have the same initiation cost removing the α term [64]. Previously, the network latency measurement was a one-way measurement between nodes, but now a two-way measurement is achieved through the use of ping-pong between i th and j th nodes with a message of fixed size [65]. The cost matrix becomes

$$T_{i,j}(m) = RTT_{i,j}(m) \tag{4.1}$$

where $T_{i,j}(m)$ is the time cost for process i to communicate to process j with a message of m bytes, $RTT_{i,j}(m)$ is the two way travel time via ping-pong to send a message between processes i and j .

We start “scanning” the node by playing ping-pong using a message size of 512 bytes. We continue to iteratively ping-pong between a set of processes until the mean round-trip time for the two processes converges. We adopted Lawrence and Yuan’s measure, requiring a 95% confidence interval to include a preset threshold of the mean, which is typically 5% of the mean [39]. A better and simpler measurement is to require the standard deviation for the measurements to be below an established threshold. We ensure that a minimum number of round-trip communications are received before the scanning phase proceeds. Once a process has completed ping-ponging with another process, it continues until it has scanned all processes within its communicator and until all processes have scanned every other process. No process will communicate with itself. The time cost for this phase is on $O(p^2)$, where p is the number of MPI processes.

The end result of the scanning phase is an $p \times p$ communication cost matrix. The cost matrix is saved and displayed and then used by an adaptive scheduling program that creates

a binomial tree ordering by searching the cost matrix for the minimum time for a particular process which corresponds to the fastest edge starting with the root process. As the algorithm determines an edge, it establishes the parent-child relationship between the selecting process and the process with the shortest time. The scheduler adds the child to the list of nodes and continues the process removing that child from the list of available processes. This continues until all nodes have been selected using the FEF criterion.

In short, this creates a new communicator where all even numbered ranks belong to one processor and all the odd numbered ranks to the other processor (Algorithm 4.7). Figure 4.5 shows an example of an FEF ordering of eight cores on two processors; the cores marked in red belong to the first processor and the cores in yellow to the second. The slowest connection is utilized in the last step of the communication.

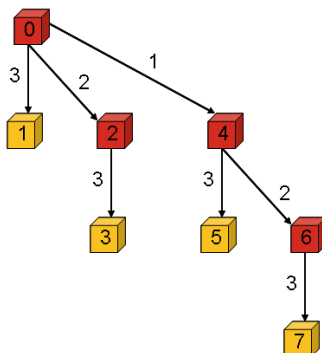


Figure 4.5: Example of an FEF ordering with two quad-core processors.

4.3 Channel Aware Ordering

Our main contribution has been the modification of the FEF heuristic to account for the multi-core network topology. On multi-core nodes, our communication model must change to reflect the fact that multi-core nodes have a highly heterogeneous, two-tiered network topology consisting of fast connections between cores on the same processor and slow connections between cores on different processors on the node [42]. We also assume that cores on multi-core nodes typically have a limited amount of network connections between

Algorithm 4.7: Fastest Edge First ordering

Input: A RTT cost matrix, rank of the Root

Output: A rank vector for the processes

```
1 size = number of processes
2 Create a RANK and PLACED vector of size
3 Create a PARENT queue
4 Initialize all RANK to 0 and PLACED to False
  /* Begin by placing the Root and setting the first RANK element to Root
  */
5 RANK [0] ← Root and PLACED [Root] ← True
6 for i ← 0 to  $\lceil \log_2(\textit{size}) \rceil$  do
  /* If a process is placed make it a parent */
7   for j ← 0 to size do
8     | if PLACED [j] = True then ENQUEUE(PARENT,i)
9   end
10  while PARENT is not empty do
11    | j ← DEQUEUE(PARENT)
12    | /* We assume that  $RTT_{j,j}$  is set to  $\infty$  */
13    | child ← j
14    | /* Find a non-placed process with the shortest  $RTT_{j,child}$  */
15    | for k ← 0 to size do
16    |   | if  $RTT_{j,k} \leq RTT_{j,child}$  and PLACED [k] = False then
17    |   |   | child ← k
18    |   end
19    |   /* Will the new rank of the child be less than or equal to size?
20    |   |   Then j should have a child */
21    |   if  $j + 2^{\lceil \log_2(\textit{size}) \rceil - \textit{step}} \leq \textit{size}$  then
22    |   |   PLACED [child] ← True
23    |   |   RANK [ $j + 2^{\lceil \log_2(\textit{size}) \rceil - \textit{step}}$ ] ← child
24    |   end
25  end
26 end
27 Display the RANK vector
```

the cores on the node. This means that it is less efficient to have cores on the same processor communicating with other cores simultaneously as their communications will interfere with one another. Note that if this assumption is not true, then there exists a better scheduling algorithm than presented.

Since there is a limited amount of network connection within a node we expect the communication cost to be higher when we try to push more messages than there are channels available. To find the amount of network connection we need to extend the ping-pong test.

The first step is to create the cost matrix from the RTT (see Equation 4.1) and the results from the matrix are then used to create two groups. The statistics for the communication times between members in the same group and between members in different groups are calculated. A t -test on the hypothesis that:

- Null: There is no significant difference between the means of the two variables.
- Alternate: There is a significant difference between the means of the two variables

In other words, $H_0 : \overline{t_{1,1}} = \overline{t_{1,2}}$ (where $\overline{t_{1,1}}$ denotes the average RTT between members in G_1 and $\overline{t_{1,2}}$ denotes the average RTT for a message from members in G_1 to members in G_2). The test is then verified with $H_0 : \overline{t_{2,2}} = \overline{t_{2,1}}$, and only if both tests reject H_0 do we accept there are two distinct groups.

Example (using values from Table 4.1):

$$G_1 = \{0, 2, 3, 5\} \qquad G_2 = \{1, 4, 6, 7\}$$

$$\overline{t_{1,1}} = 30.63 \qquad s_{1,1} = 0.152 \qquad n = 12$$

$$\overline{t_{1,2}} = 32.77 \qquad s_{1,2} = 0.167 \qquad n = 16$$

$$H_0 : \overline{t_{1,1}} = \overline{t_{1,2}} \qquad p = 2.5 \cdot 10^{-23}$$

$$\begin{aligned} \overline{t_{2,2}} &= 30.53 & s_{2,2} &= 0.092 & n &= 12 \\ \overline{t_{2,1}} &= 32.83 & s_{2,1} &= 0.326 & n &= 16 \end{aligned}$$

$$H_0 : \overline{t_{2,2}} = \overline{t_{2,1}} \quad p = 4.1 \cdot 10^{-19}$$

This confirms the first criterion; that we have two groups with different bandwidth between members within the group and members outside the group. The second criterion for the Channel Aware Ordering (CAO) is that it should be possible to send two or more messages in parallel between groups without a significant loss in bandwidth. To test this, we have to extend the ping-pong to measure RTT for parallel messages (Figure 4.6). This is done in three steps:

1. Measure the parallel RTT within a single group alone.
2. Measure the parallel RTT within a single group simultaneously within the two groups.
3. Measure the parallel RTT between the two groups.

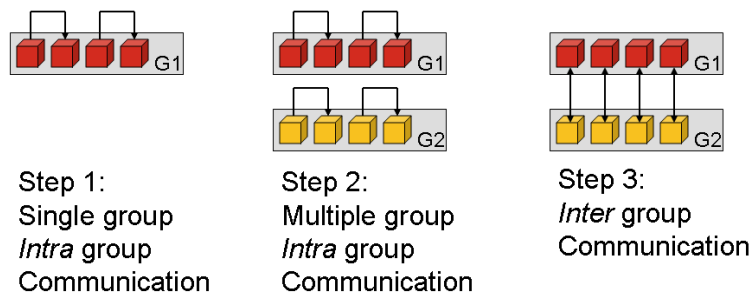


Figure 4.6: Example of Extended Parallel Ping-Pong Tests, with two quad-core processors.

After determining how many messages can be passed in parallel between the two processors without any significant loss in bandwidth, the algorithm creates a communicator with the cores ordered in such a way that messages will cross between the two processors when the optimum number of parallel messages have been reached. This means that if more than a single message can be passed without any loss the first step will be communication between

two cores on the same processor. The next step is to order the process such that we cross the boundary between the two groups when the number of channels are reached (Algorithm 4.8). Figure 4.7 shows an example of CAO ordering of two processors with four cores, able to pass two messages in parallel between the two processors. The red cores belong to processor one and the yellow to the second processor.

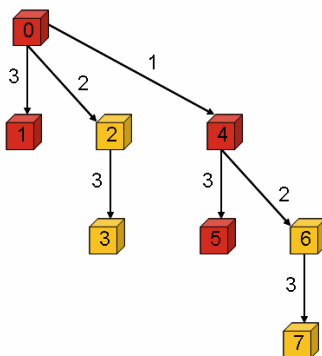


Figure 4.7: Example of an CAO ordering with two quad-core processors.

4.4 Random Ordering

After the adaptive trees are constructed another scheduling program creates a naive binomial tree schedule. The function starts with the root node and creates a random permutation of a sorted array (Algorithm 4.9). This schedule is used to isolate the effect of the FEF and CAO heuristic from the effect of using any binomial tree.

4.5 Re-Mapping

Just as MPI methods were developed prior to the introduction of multi-core systems, so were the methods of organizing their computational processes into multidimensional grids [21]. Testing has revealed that today’s state-of-the-art MPI implementations are often sub-optimal for multidimensional communication because the standard mapping `MPI_Init()` and `MPI_Cart_create()` does not take into account how many times a specific message is passed between different parts of the hardware, nor does it consider how many messages

Algorithm 4.8: Channel Aware Ordering ordering

Input: A RTT cost matrix, number of channels and rank of the Root

Output: A rank vector for the processes

```
1 size = number of processes
2 Create a RANK and PLACED vector of size
3 Create a PARENT queue
4 Initialize all RANK to 0 and PLACED to False
5 RANK [0]  $\leftarrow$  Root and PLACED [Root]  $\leftarrow$  True
6 for  $i \leftarrow 0$  to  $\lceil \log_2(\textit{size}) \rceil$  do
7   for  $j \leftarrow 0$  to size do
8     | if PLACED [ $j$ ] = True then ENQUEUE(PARENT, $i$ )
9   end
10  while PARENT is not empty do
11    |  $j \leftarrow$  DEQUEUE(PARENT)
12    | /* Set the parent to its own child */
13    | child =  $j$ 
14    | if  $i = \textit{channels}$  then  $RTT_{j,\textit{child}} \leftarrow 0$ 
15    | else  $RTT_{j,\textit{child}} \leftarrow \infty$ 
16    | for  $k = 0$  to size do
17    |   | if  $i = \textit{channels}$  then
18    |     | /* Find a member from the other group */
19    |     | if  $RTT_{j,k} \geq RTT_{j,\textit{child}}$  and PLACED [ $k$ ] = False then
20    |     |   | child  $\leftarrow k$ 
21    |     | end
22    |     | else
23    |     |   | /* Find a non-placed process with the shortest  $RTT_{j,\textit{child}}$  */
24    |     |   | if  $RTT_{j,k} \leq RTT_{j,\textit{child}}$  and PLACED [ $k$ ] = False then
25    |     |     | child  $\leftarrow k$ 
26    |     |   | end
27    |     | end
28    |     | if  $j + 2^{\lceil \log_2(\textit{size}) \rceil - \textit{step}} \leq \textit{size}$  then
29    |     |   | PLACED [child]  $\leftarrow$  True
30    |     |   | RANK [ $j + 2^{\lceil \log_2(\textit{size}) \rceil - \textit{step}}$ ]  $\leftarrow$  child
31    |     |   | end
32    |     | end
33    |     | Display the RANK vector
```

Algorithm 4.9: Random ordering

Input: rank of the Root and number of processes
Output: A rank vector for the processes

- 1 $size \leftarrow$ number of processes
- 2 Create a RANK vector of $size$
 /* Initialize all RANK */
- 3 **for** $i \leftarrow 0$ to $size$ **do** RANK [i] $\leftarrow i$
- 4 RANK [0] \leftrightarrow RANK [$Root$]
- 5 **for** $i \leftarrow 1$ to $size - 1$ **do**
- 6 | $rndpos \leftarrow$ Random number between i and $size$
- 7 | RANK [i] \leftrightarrow RANK [$rndpos$]
- 8 **end**
- 9 Display the RANK vector

can be passed simultaneously over different hardware connections [49–52, 66]. We still want to use the MPI methods for message passing without making any changes to the methods themselves. To solve this we have to find a method that allows us to change the process-to-core mapping so we can utilize the high performing MPI message passing protocols. We have developed a method that we call *Re-mapping* that allows us to do just that.

The `MPI_Init()` creates an automatic mapping from a physical core to a process [16, 47, 48]. In a worst case scenario, this may be a random mapping where we can assume $(n \cdot k)!$ different permutations, where n is the number of nodes and k the number of cores on each node. Yet, we know that this is not true and that most implementations of MPI actually provide for four different rank re-order methods (Figure 4.8):

- **Round-robin:** Sequential MPI ranks are placed on the next node in the list.
- **SMP-style:** All cores from all nodes are allocated in a sequential order.
- **Folded rank:** Similar to the first ordering except that the tasks $n + 1 \dots 2n$ are mapped to slave cores of nodes $n \dots 1$.
- **Custom ordering:** The ordering is specified in a file.

Round-Robin mapping creates a process rank with k chunks, each with n cores from different nodes arranged after each other. *SMP-style* mapping produces a process rank with

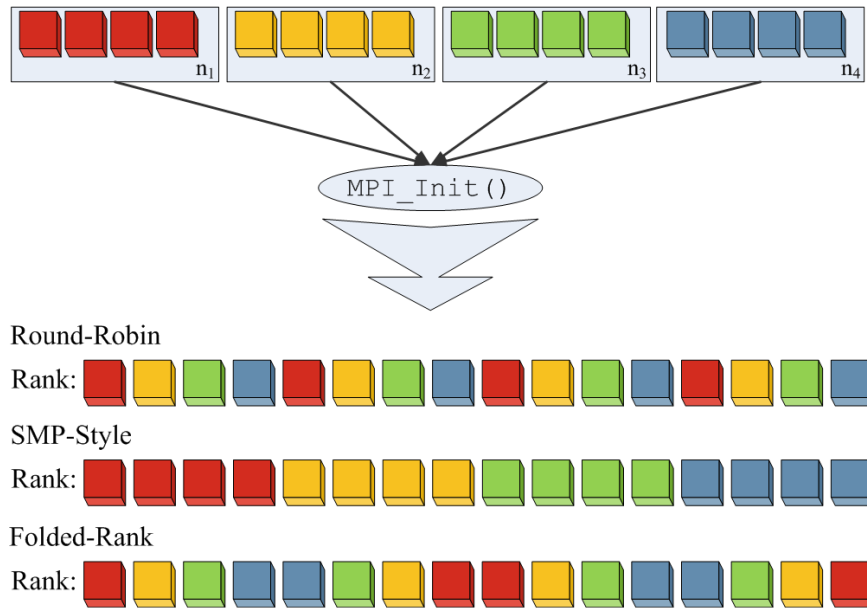


Figure 4.8: Example of a Process to Core Mapping, four nodes with four cores each.

n chunks of k cores all from the same node, one after another. Finally, *Folded rank* mapping creates a process rank consisting of k chunks with n cores from different nodes similar to the first one, but the node order is reversed after each chunk.

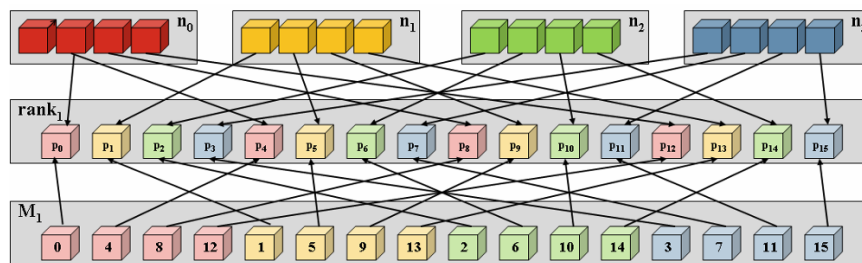
The `MPI Communicator` defines the communication domain where it, among other things, defines the set of processes that can be contacted. Each such process is labeled by a process rank: a set of integers that can be discovered by `MPI_Comm_rank()`. Assuming one process per physical core, the number of processes will equal $p = n \cdot k$. The `MPI_Comm_rank()` gives these processes an MPI internal numbering indexing them from 0 to $p - 1$, where we let $p[i]$ denote the i^{th} process.

As mentioned before, this mapping from a physical core to a process is done automatically through the `MPI_Init()` (see Figure 4.8). `MPI_Init()` and `MPI_Cart_create()` have typically been the methods of creating effective mappings, but they do not consider how many times a single message must make point-to-point calls between nodes or in the single node case how many point-to-point are made between the different hardware chips on the node. Testing has since revealed that this lack of consideration indicates that the process-to-core mapping and

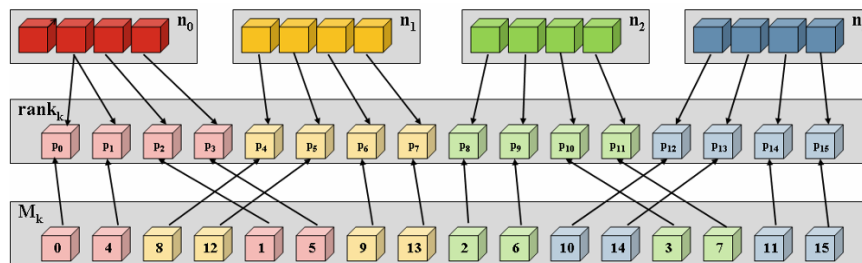
rank ordering created by `MPI_Init()` and `MPI_Cart_create()` are many times sub-optimal. The purpose of our system is to redistribute the processes onto cores in order to achieve optimal performance for a given algorithm.

Definition 1 *Re-Mapping*

Let M_i be a vector defined on a given communicator *The MPI Communicator* s.t. it is a mapping from the cores given by the **rank** on that communicator to a different core in a new rank, where m_i indicates the core in the original **rank**



(a) M_1 : Round Robin \rightarrow SMP-Style Re-mapping



(b) M_k : SMP-style \rightarrow Tiled Rank Re-Mapping

Figure 4.9: Example of two different Re-mappings; $n = 4$, $k = 4$

The re-mapping provides us with a method and `MPI_Comm_group()`, `MPI_Group_incl()` and `MPI_Comm_create()` the tools to create a new MPI communicator, using information about the underlying structure of the hardware. We can analyze the rank-ordering and then re-map the order to better suit our purpose (see Figure 4.9). Figure 4.9(a) shows an example where re-mapping the given *Round Robin* ordering into an *SMP* ordering and Figure 4.9(b) provides an example of re-mapping an *SMP* ordering into a custom ordering,

in this case called Tiled Rank. By letting our mapping M_i be the ordered input of ranks for the `MPI_Group_incl()` we can create a new group derived from the `MPI_COMM_WORLD` communicator.

This is the function description for `MPI_Group_incl()`:

The function `MPI_Group_incl()` creates a group *newgroup* that consists of the p processes in *group* with ranks $rank[0], \dots, [p-1]$; the process with rank i in *newgroup* is the process with rank $rank[i]$ in *group*. Each of the p elements of ranks must be a valid rank in *group* and all elements must be distinct, or else the call is erroneous. If $p = 0$, then *newgroup* is `MPI_GROUP_EMPTY`. `MPI_GROUP_EMPTY` This function can, for instance, be used to reorder the elements of a group [21].

Let us look at an example: The cost matrix of the RRT from the RA cluster told us there were two distinct groups within a single node (Table 4.1). With ranks $\{0, 2, 3, 5\}$ in *Group1* and ranks $\{1, 4, 6, 7\}$ in *Group2*. We cannot exactly determine which core is mapped to which process but we do have a fairly good picture (Figure 4.10).

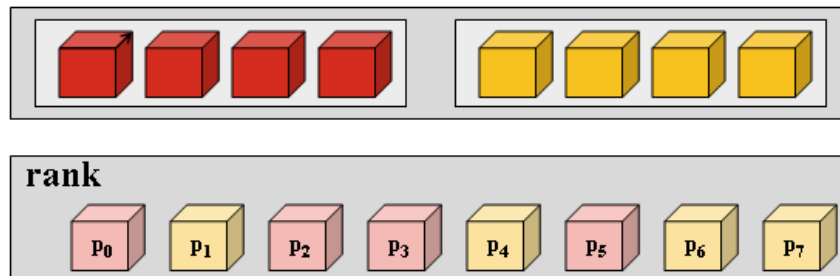
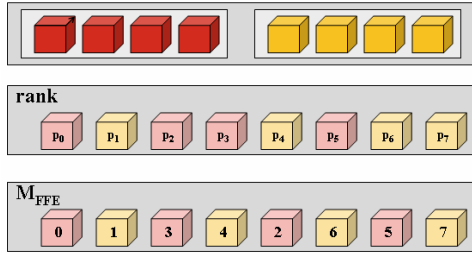


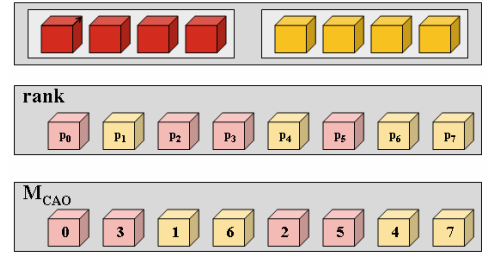
Figure 4.10: Example of the rank of single node on the RA cluster, two distinct groups with four cores in each.

The Fastest Edge First Algorithm (4.7) will return a RANK vector $RANK = \{0, 1, 3, 4, 2, 6, 5, 7\}$, (see Figure 4.11(a)). This rank-vector will correspond to our Re-map vector M_{FFE} .

We could also apply the Channel Aware Ordering Algorithm (4.8), that would return $RANK = \{0, 3, 1, 6, 2, 5, 4, 7\}$, (see Figure 4.11(b)). This will correspond to our Re-map vector M_{CAO} .



(a) Re-mapping using the Fastest Edge First Algorithm



(b) Re-mapping using the Channel Aware Ordering

Figure 4.11: Re-mapping of the original Rank on RA

Algorithm 4.10: Create Communicator

Input: Re-map vector and MPI_COMMUNICATOR

Output: A re-ordered MPI_COMMUNICATOR

- 1 $size \leftarrow$ number of processes
/* Access the GROUP from the underlying MPI_COMM_WORLD */
 - 2 $group \leftarrow$ MPI_Comm_group()
/* Create a NEWGROUP from the GROUP and the Re-map vector */
 - 3 $NEWGROUP \leftarrow$ MPI_Group_incl()
/* Create a new communicator from the NEWGROUP */
 - 4 $MPI_COMMUNICATOR \leftarrow$ MPI_Comm_create()
 - 5 Display and return the new communicator
-

Our last step is the create new communicators on the re-map vectors. We can use these newly created communicators in the standard MPI message passing protocols. With the advantage that our processes are ranked in a preferable ordering, we can optimize the message passing without making any changes to any of the standard MPI functions and protocols.

The groups for the new *FFE*-communicator will be:

$$G_{red} = \{0, 2, 4, 6\}$$

and

$$G_{yellow} = \{1, 3, 5, 7\}$$

when dispersing an $8 \cdot m$ message using `MPI.Scatter()` on the *FFE*-communicator the two first steps are communication between members of the within the same group (G_{red}) and the message will be crossing boundary between the groups in the last step (see Table 4.4):

Table 4.4: A message of $8 \cdot m$, scattered using the *FFE*-communicator.

Step	From \rightarrow To	Group	Message size
1	$p_{[0]} \rightarrow p_{[4]}$	red \rightarrow red	$4 \cdot m$
2	$p_{[0]} \rightarrow p_{[2]}$	red \rightarrow red	$2 \cdot m$
	$p_{[4]} \rightarrow p_{[6]}$	red \rightarrow red	$2 \cdot m$
3	$p_{[0]} \rightarrow p_{[1]}$	red \rightarrow yellow	m
	$p_{[2]} \rightarrow p_{[3]}$	red \rightarrow yellow	m
	$p_{[4]} \rightarrow p_{[5]}$	red \rightarrow yellow	m
	$p_{[6]} \rightarrow p_{[7]}$	red \rightarrow yellow	m

and for the new *CAO*-communicator:

$$G_{red} = \{0, 1, 4, 5\}$$

and

$$G_{yellow} = \{2, 3, 6, 7\}.$$

the communication pattern for an $8 \cdot m$ message on the *CAO*-communicator will be (see Table 4.5):

We also have the ability to create any ordering we so desire and apply it to test any of the standard MPI functions and protocols. The *Re-mapping* algorithm provides us with the

Table 4.5: A message of 8 m , scattered using the *FFE*-communicator.

Step	From \rightarrow To	Group	Message size
1	$p_{[0]} \rightarrow p_{[4]}$	red \rightarrow red	$4 \cdot m$
2	$p_{[0]} \rightarrow p_{[2]}$	red \rightarrow yellow	$2 \cdot m$
	$p_{[4]} \rightarrow p_{[6]}$	red \rightarrow yellow	$2 \cdot m$
3	$p_{[0]} \rightarrow p_{[1]}$	red \rightarrow red	m
	$p_{[2]} \rightarrow p_{[3]}$	yellow \rightarrow yellow	m
	$p_{[4]} \rightarrow p_{[5]}$	red \rightarrow red	m
	$p_{[6]} \rightarrow p_{[7]}$	yellow \rightarrow yellow	m

necessary tool to define an optimization function as:

$$\min [T(\mathbb{M}_0), T(\mathbb{M}_1), \dots, T(\mathbb{M}_{\|\mathbb{M}\|})] \quad (4.2)$$

where $T(\mathbb{M}_i)$ is the communication time implemented using the i^{th} mapping.

This re-mapping has to be done every time we run the program; the only things we will avoid after the first execution are the steps to confirm that we have two distinct groups and to determine the amount of channels available for parallel communication within and between the groups. Unlike the case when `MPI_Init()` maps several cores on several nodes rank list, where we know from which node a specific rank will be mapped, we have no control over which specific core will be placed at which rank. This means we cannot make any assumptions in regard to the ordering of cores from different hardware chips within the same node.

4.6 Evaluation

With the introduction of *Re-mapping* using FEF and CAO we can conduct tests and experiments and compare the results to both a randomized binomial tree and the default implementation provided through the MPI-library. Three different systems were used for these experiment; *RA*, *Jaguar* and *BlueDrop* (see Chapter 3.7.1 for specifications).

4.6.1 Theoretical Assumptions

Assuming a system with two quad-core processors, and binomial tree implementation of a scatter function where each core receives a message of size m . Further, there is a difference in bandwidth for communication between cores on the same processor (β_1) and communication between cores on different processors (β_2). The connection between each processor allows two messages to be sent in parallel both between cores on the same processor and between cores located on different processors without any loss in bandwidth. We can estimate the expected time to scatter the message to (see Table 4.6):

Table 4.6: The cost to scatter a message of size $8m$ to eight cores in two groups, allowing two messages to be sent in parallel.

Algorithm	Step 1	Step 2	Step 3	Total
Standard Ordering	$4m\beta_2$	$2m\beta_1$	$m\beta_1$	$3m\beta_1 + 4m\beta_2$
FEF	$4m\beta_1$	$2m\beta_1$	$(4/2)m\beta_2$	$6m\beta_1 + 2m\beta_2$
CAO	$4m\beta_1$	$2m\beta_2$	$m\beta_1$	$5m\beta_1 + 2m\beta_2$

The standard ordering assumes that the communicator is arranged in such a way that all the cores from processor one are first in rank and the cores from the second processor follows in rank. We can quickly conclude that if it is possible to send two messages in parallel between the processors the CAO algorithm should be less costly than the FEF, and if $\beta_2 > 1.5 \cdot \beta_1$ FEF would be faster than the standard ordering.

4.6.2 Experiments and Results

Our first test was to create the cost matrix to confirm that each system has two groups with different bandwidth between members within the group and members outside the group (see Table 4.7).

The results and t-test confirms that all three systems have two distinct groupings and that there is an opportunity for both FEF and CAO to improve the standard implementation.

The second pre-test was the extended ping-pong test, to verify how many messages can be sent between the two groups without any loss in bandwidth.

Table 4.7: Cost Matrix results (round trip travel-times) for all three systems.

System	Time (μsec)	Stddev (μsec)	n
RA			
$t_{1,1}$	30.63	0.152	12
$t_{1,2}$	32.77	0.167	16
$t_{2,2}$	30.53	0.092	12
$t_{2,1}$	32.83	0.326	16
Kraken			
$t_{1,1}$	41.75	0.846	30
$t_{1,2}$	45.61	0.802	36
$t_{2,2}$	41.38	0.833	30
$t_{2,1}$	45.04	0.815	36
BlueDrop			
$t_{1,1}$	35.15	0.347	240
$t_{1,2}$	59.88	0.711	256
$t_{2,2}$	34.25	0.352	240
$t_{2,1}$	56.72	0.541	256

Table 4.8: Extended Ping-Pong results for all three systems.

System	Time (MB/sec)	Stddev (MB/sec)
RA		
1 \leftrightarrow 1	1228	65
2 \leftrightarrow 2	716	52
3 \leftrightarrow 3	544	60
4 \leftrightarrow 4	382	58
Kraken		
1 \leftrightarrow 1	2113	59
2 \leftrightarrow 2	1104	63
3 \leftrightarrow 3	987	77
4 \leftrightarrow 4	909	64
5 \leftrightarrow 5	878	58
6 \leftrightarrow 6	880	103
BlueDrop		
1 \leftrightarrow 1	2220	72
2 \leftrightarrow 2	2153	87
3 \leftrightarrow 3	1212	68
4 \leftrightarrow 4	1187	76
5 \leftrightarrow 5	1123	62
6 \leftrightarrow 6	1098	69
...

The results from both RA and Kraken (see Table 4.8) show a significant drop in bandwidth when trying to pass more than a single message in parallel. It should mention that the bandwidth is calculated on the slowest message in the group, so in the case of RA, two messages were passed and the slowest bandwidth was measured at 716 MB/sec. Previous tests using an averaging method showed less of a drop in the bandwidth at two messages. Our assumption is that one channel is still able to perform at full capacity. We therefore decided to include both RA and Kraken in further tests.

We ran four different tests on each system. We first ran a scatter test with a message size of 10 MB for each core using the default MPI-ordering. Next, a random ordering was created and a scatter test with the same message on this communicator was run. These two tests concluded our baseline data. The third test was our FEF algorithm, and the fourth the CAO rank; both of these used the same message as the two baseline tests. We ran each test 20 times on all systems to get a significant sample size (n) for the analysis.

The results from RA shows no significant improvement (see Table 4.9). There is an increase in communication time when using the random communicator, but it has no statistical significance. The time (10.95 *msec*) it takes to create the FEF and CAO makes them even slower than the standard MPI-ordering. The pre-test showed that there is a small, but still significant, difference in bandwidth when exchanging messages within or between the processors. This difference is not large enough to have any impact on the FEF or CAO algorithm. Further, that inability to pass more than a single message in parallel between the two processors prevented any improvement from the CAO algorithm.

Table 4.9: The results from RA, $n = 20$ and 10 MB message size.

Communicator	Time (msec)	Stddev (msec)
Standard Rank	308.62	11.00
Random Rank	313.66	11.16
FEF Rank	308.90	11.53
CAO Rank	308.25	11.22

Kraken showed a statistically significant improvement between the FEF and CAO algorithm, confirming that it might be beneficiary to consider when to pass the message over the border between the two processors (see Table 4.10). A noticeable fact is that Kraken is the only system of the three where the binomial tree is constructed from an amount of cores that is not a power of two (twelve). The overhead to create the FEF and CAO communicators are 12 *msec* so it could therefore be cost-efficient to reorder the communicator even for small amounts of messages.

Table 4.10: The results from Kraken, $n = 20$ and 10 MB message size.

Communicator	Time (msec)	Stddev (msec)
Standard Rank	104.36	5.79
Random Rank	106.80	6.42
FEF Rank	101.22	8.04
CAO Rank	80.94	7.64

Finally, the results from BlueDrop shows that both FEF and CAO are confirmed faster than the standard implementation, and the difference between them is statistically significant (see Table 4.11). The noticeable difference between the standard rank and the reordering confirms it is a good idea to consider reordering if there is a significant amount of messages. There is a considerable overhead ($t = 60.04$ *msec*) so the amount or the size of the messages have to be large to make it worthwhile to re-order the communicator.

Table 4.11: The results from BlueDrop, $n = 20$ and 10 MB message size.

Communicator	Time (msec)	Stddev (msec)
Standard Rank	123.41	3.08
Random Rank	126.11	3.56
FEF Rank	116.10	3.25
CAO Rank	111.76	2.75

We broke down the communication of the BlueDrop into each individual step, to be able to analyze each step (see Table 4.12). The pre-tests confirmed the existence of two distinct groupings, and that two messages can be sent in parallel between cores in each group, and

two messages can be shared simultaneously within each group. If we assume that the average time for each individual step are independent random variables we are able to make some statistical analysis of the results.

Table 4.12: The step by step results from BlueDrop, $n = 20$ and 10 MB message size.

Algorithm/Step	Time (msec)	Stddev (msec)
Standard Rank		
Step 1	92.26	0.48
Step 2	46.39	0.19
Step 3	27.76	0.09
Step 4	14.98	0.43
Step 5	6.29	0.36
Total:	187.68	0.78
FEF		
Step 1	50.70	0.55
Step 2	36.76	0.11
Step 3	19.75	0.56
Step 4	14.38	0.41
Step 5	11.51	0.08
Total:	133.10	0.91
CAO		
Step 1	50.13	0.59
Step 2	45.53	0.13
Step 3	18.68	0.60
Step 4	10.47	0.27
Step 5	6.43	0.05
Total:	131.24	0.93

Both FEF and CAO are significantly faster than the Standard rank. There is also a noticeable expected difference between the FEF and CAO in the second and the last step. The difference in the fourth step is hard to explain, and it has an impact on the overall performance.

CHAPTER 5

MULTI-DIMENSIONAL MPI COMMUNICATIONS

In this chapter we introduce an algorithm to improve the multidimensional communication time by as much as $\Theta(\frac{1}{d}ck^{\frac{d-1}{d}})$ over the default mapping, where c is the number of the network ports per node and k is the number of the computational cores per node and d the number of dimensions in the process-grid.

What do we mean by multidimensional communication? The simple answer is: the type of communication where the processes communicate with the processes around them, or with all the processes that are located along the same axis (dimension). This sort of communication means that the processes have to be organized into a multidimensional process grid. The user usually does this by either invoking the `MPI_Cart_create()`, or by simply defining a set amount of rows (r) and columns (c), and then letting $myrow = myrank/c$ and $mycolumn = myrank/r$, etc. Independent of what method the user applied, each process now has a *rank*, *row*, *column* etc. where no two processes share the same parameter value. We can now define a process by either its *rank* or by its location in the defined process grid, and there is a subsequent correlation between the *rank* and position [67].

The multidimensional communication between the processes can be *point-to-point* or *collective*. In the *point-to-point* method each process sends and receives messages from its neighbor around it (above, below, right, left, et cetera). In the *collective communication* the application uses operations such as *All-to-All*, *All-to-One* or *One-to-All*, to pass the data.

5.1 Background

Many applications already organize their processes in multidimensional Cartesian grids wherein communications often need to be performed in each dimension simultaneously. This is the basis for multidimensional communication. This is uniquely different from standard MPI operations which are optimized in regards to a single axis of the complete Cartesian

grid (see Figure 5.1). The multidimensional communication employs all the available axes of the Cartesian grid simultaneously, as opposed to utilizing only a single axis. In addition to this, the messages being passed along these multiple axes will not necessarily contain the same information. Thus, multidimensional communication must account for all these variables and continue to perform efficiently during each step of the process.

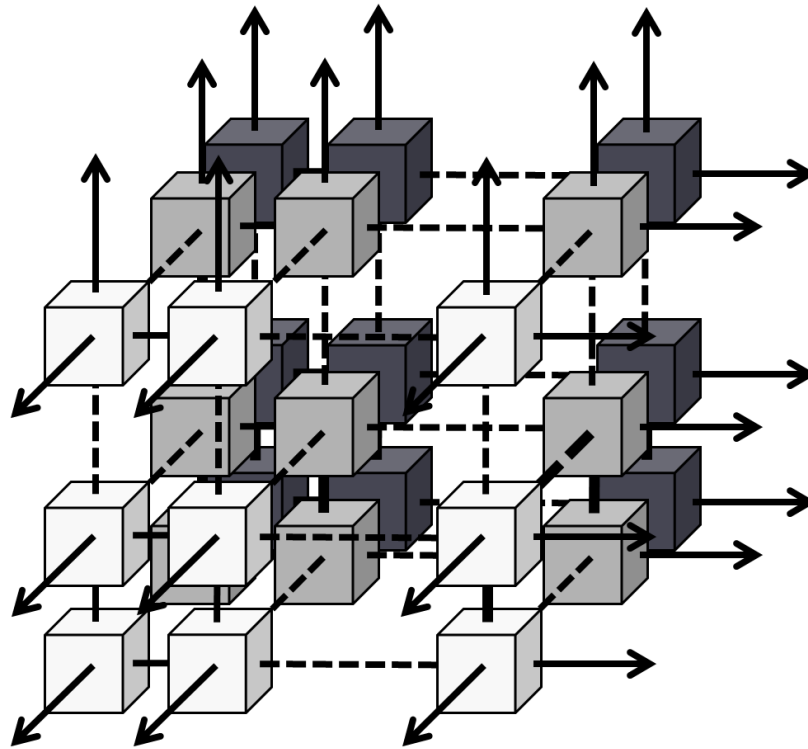


Figure 5.1: An example of a 3 dimensional process grid.

For example, at every iteration in Matrix-Matrix Multiplication sub-matrices are broadcast simultaneously along both the *rows* and the *columns*. This is a good representation of 2D `MPI_Bcast()`, the relevant section with respect to multidimensional communication is shown in the source code below Figure 5.2. This operation is defined by first creating two communicators (*row* and *column*), by using either `MPI_Comm_split()` or `MPI_Cart_create()`. In the next step, a message (not necessarily the same message) is then passed by using `MPI_Bcast()`. The message is then broadcast along both the row and column

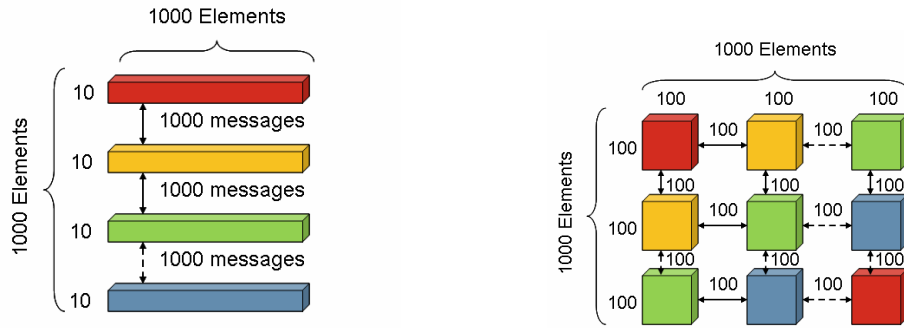
during each iteration. Neither the Cartesian grid nor the *row* and *column* communicators are changed once they have been created.

```
/* Create the row communicator */
MPI_Comm_split(world_comm, my_row,
               myproc, my_row_comm);
/* Create the column communicator */
MPI_Comm_split(world_comm, my_col,
               myproc, my_col_comm);
...
/* Broadcast the row message */
MPI_Bcast(row_message, size,
          MPI_TYPE, root, my_row_comm);
/* Broadcast the column message */
MPI_Bcast(col_message, size,
          MPI_TYPE, root, my_col_comm);
```

Figure 5.2: Example of 2D `MPI_Bcast()`, two sequential broadcasts using two different communicators.

By this definition, if the dimensions of the process grid would be three then the multidimensional communication would be denoted `3D MPI_Bcast()`. Further, this definition is not limited to `MPI_Bcast()` operations; the latter part can be used to denote what MPI operation is being employed. For example, using *gather* or *scatter* over a three dimensional grid would be called `3D MPI_Gather()` or `3D MPI_Scatter()`.

There are several reasons why it is important to consider the dimensionality of underlying Cartesian grid. First, to be as efficient as possible when we create a parallel code we often try to arrange our process-grid so it matches the structure and dimension of the data. For example when we try to solve a Partial Differential Equation (PDE) of 1000×1000 grid elements on 100 processes. We could arrange the processes such that each process handles ten rows of grid-elements, or we let each process handle a block of 100×100 grid-elements (see Figure 5.3).



(a) Grid decomposed in one dimension, 10000 grid elements per process. Each process sends $2 \cdot 1000$ messages each iteration.

(b) Grid decomposed in two dimensions, 10000 grid elements per process. Each process sends $4 \cdot 100$ messages each iteration.

Figure 5.3: An example of a PDE: 1000x1000 grid-elements solved on 100 processes.

If we elect to use the first arrangement, the amount of messages passes in each step of the calculation will amount to (Figure 5.3(a)):

$$2 \cdot 1000 \text{ messages/process}$$

While if we would arrange the processes to better reflect the dimensional structure of our data, the amount of messages passed each step would be (Figure 5.3(b)):

$$4 \cdot 100 \text{ messages/process}$$

Both of these estimations are only considering the messages passed in the interior of the grid.

Second, these types of communication operations are not found within the standard MPI library. Thus, they have been neglected in terms of optimization tuning in regards to the needs of modern applications. This means that the performance of these combined MPI functions are often sub-optimal in multi-dimensional scenarios, especially when compared to their effectiveness along a single axis. The purpose of this part of the research is to address this oversight by focusing the efforts on multidimensional *Collective communications*. We will use common operations (such as the standard MPI) as tools for an application-level solution and focus on creating communicators that will optimize the combined operations.

To be able to analyze the system we will use a simple model for the parallel computing, with the following assumptions:

- **Indexing:** Assuming one process per physical core, the parallel architecture consists of n nodes with k cores per node, giving a total of $n \cdot k$ computational processors. With physical cores indexed and named, it is possible to obtain part of this information using `MPI_Get_processor_name()`. The `MPI_Comm_rank()` assigns the processes a MPI internal numbering, indexed from 0 to $p-1$, where $p[i]$ denotes the i^{th} process.
- **Logically fully connected:** Any process can send directly to any other process where a communication network provides automatic routing.
- **Locality:** The communication between two processes are described by their relative location. If $p[i]$ and $p[j]$ are located on the same node, the communication will be denoted *Internal* and I will be used as the subscript (*intra-communication*). If $p[i]$ and $p[j]$ are located on different nodes the communication will be denoted *External* and E will be used as the subscript (*inter-communication*). Example:
 - *Internal* latency: α_I
 - *External* latency: α_E
 - *Internal* bandwidth: $1/\beta_I$
 - *External* bandwidth: $1/\beta_E$
- **Cost of communication:** The cost of sending a message of size m between process $p[i]$ and $p[j]$ will be modeled by $T_{i,j}(m) = \alpha + m \cdot \beta$, in the absence of network conflicts. Here α denotes the message startup time, β the data transmission time, where $\frac{1}{\beta}$ is the network bandwidth.

We assume that the *Internal* communication between processes are all equal.

$$\begin{aligned}
T_{0,1}(m) &= T_{0,2}(m) = \dots = T_{0,k-1}(m) \\
T_{ik,ik+1}(m) &= T_{ik,ik+2}(m) = \dots = T_{ik,k(i+1)-1}(m) \\
T_{0,1}(m) &= T_{k(n-1),nk-1}(m)
\end{aligned}$$

Further, we will also assume that *External* communication between nodes are equal.

$$\begin{aligned}
T_{0,k}(m) &= T_{0,k+1}(m) = \dots = T_{0,2k-1}(m) \\
T_{1,k}(m) &= T_{1,k+1}(m) = \dots = T_{1,2k-1}(m) \\
&\dots \quad \dots \\
T_{ik,k(n-1)}(m) &= T_{ik,k(n-1)+1}(m) = \dots = T_{ik,nk-1}(m) \\
&\dots \quad \dots \\
T_{0,k} &= T_{1,k}(m) = \dots = T_{k(n-1)-1,nk-1}(m)
\end{aligned}$$

- **Cost of computation:** The cost required to perform an arithmetic operation will be denoted by γ . We assume a homogeneous cluster and claim that $\gamma_{p[i]s} = \gamma_{p[j]s}$.

5.2 Tiling

Most often, an application will perform very well along a single axis of the Cartesian grid, but once it attempts to work and communicate along a secondary (or even third) axis the application's efficiency suffers notably. This is due to the fact that shared memory is leveraged towards exchanging messages within nodes rather than making point-to-point calls between the nodes. The communication stages within the nodes then take advantage of not moving data across the network and therefore minimizing contention. The grids in Figure 5.4 though small, show that if the processes are arranged in a *Round-Robin* fashion each node of cores (boxes of the same color) would have 4 *Inter*-communications (communication between different nodes) along the x -axis and 3 *Intra*-communications (communication within the same node) along the y -axis (Figure 5.4(a)). If the processes would be arranged according to *SMP-style* the communication in each step would instead be 3 *Intra*-communications along the x -axis and 4 *Inter*-communications along the y -axis (Figure 5.4(b)).

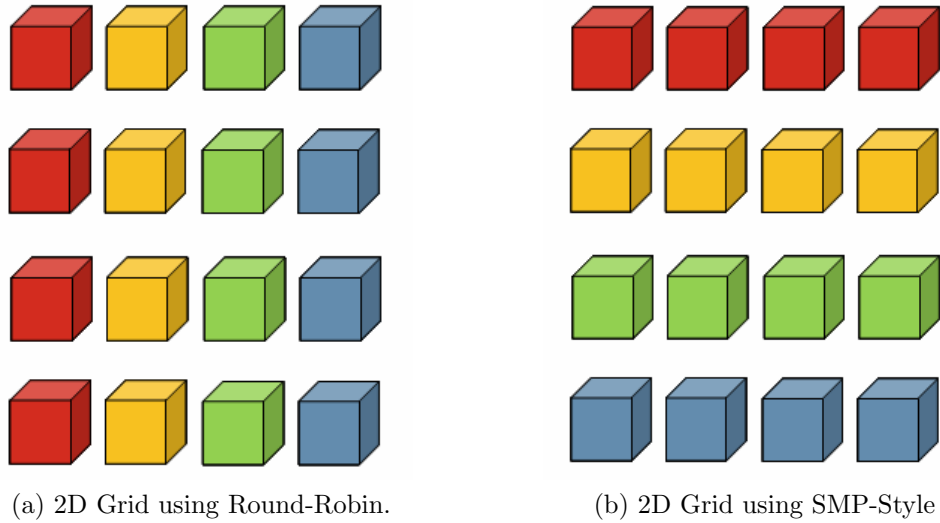


Figure 5.4: An example of the Inter vs. Intra Communication on small grids using four nodes with four cores each.

In this case the total communication cost is independent of the algorithm used to arrange the processes into *rank* during the initialization of the MPI, but on larger grids this is not the case necessarily. We propose using a tiling method to optimize the overall performance of the application rather than only its operations in a single direction. This tiling method would ensure that the total amount of *Inter*-communications over the complete grid would be kept to a minimum.

Instead of viewing the rank as an array of processes, the processes on each node should be viewed and interpreted as a tile (see Figure 5.5). The shape of these tiles is dependent not only on the type of communication but also on the size and shape of the grid in which the communication will occur as well as the size and distribution of the different messages. The goal is to develop an algorithm that automatically re-maps the processes and provide an optimal communicator once given the information regarding both the size and structure of the grid and the number of processes involved. The implementation automatically determines the number of cores per node and finds an optimal shape for a single tile.

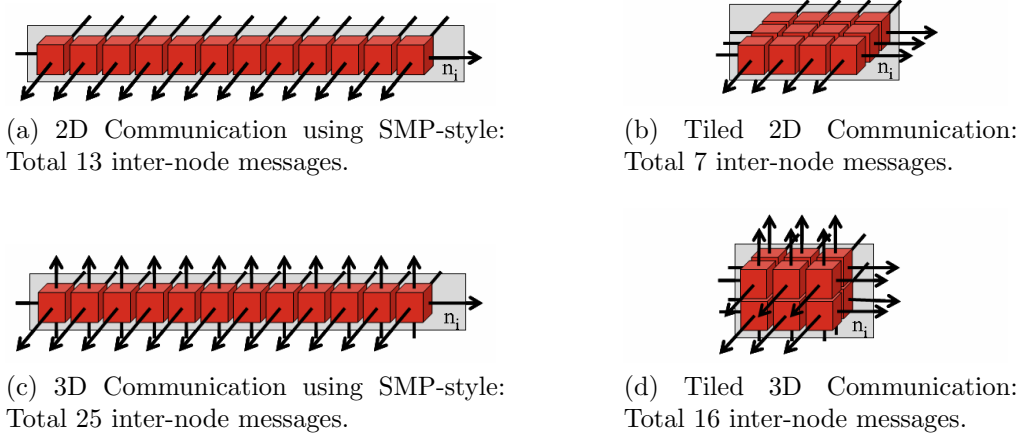


Figure 5.5: An example of inter-node messages on a twelve core node using different tiles.

5.2.1 Node mapping

To enable us to find the best mapping and tiling for our purpose, we need to know on which node a given process is located. We will therefore introduce **Node Mapping**. First, create a list of all possible nodes by the use of `MPI_Get_processor_name()`. Second, from this list assign an integer $node = 0 \dots n$ to each process depending on which physical node it is located. Third, create a vector \mathbb{N} s.t. \mathbf{n}_i is the integer for the node that process p_i is located on.

Definition 2 *Node Map* Let, $f_{i,d}(\mathbb{N}_j)$ denote the number of outgoing communications from node i over the d^{th} -dimension in mapping \mathbb{M}_j :

$$f_{i,d}(\mathbb{N}_j) = \sum_{m=0}^{(n \cdot k) - 1} a_m \text{ where } a_m \begin{cases} 0 & \text{if } \mathbf{n}_m = \mathbf{n}_{m+1} \\ 1 & \text{if } \mathbf{n}_m = i \neq \mathbf{n}_{dm+1} \end{cases} \quad (5.1)$$

if $\mathbf{n}_m, \mathbf{n}_{m+1}$ are in same dimension d

Example: Two different mappings of four nodes ($n = 4$) each with four cores ($k = 4$) for a total of 16 processes ($p = 16$) (see Figure 5.6). In the standard mapping (\mathbb{M}_{std} see Figure 5.6(a)) none of the processes communicate with any other node when the message is passed along the x-axis (d_1), but for the y-axis (d_2) all processes in three nodes $n = \{0, 1, 2\}$

has to be pass the message to another node. In the tiled mapping (\mathbb{M}_{tiled} see Figure 5.6(b)) there are two by two external communications in the x-axis (d_1) and four in the y-axis (d_2).

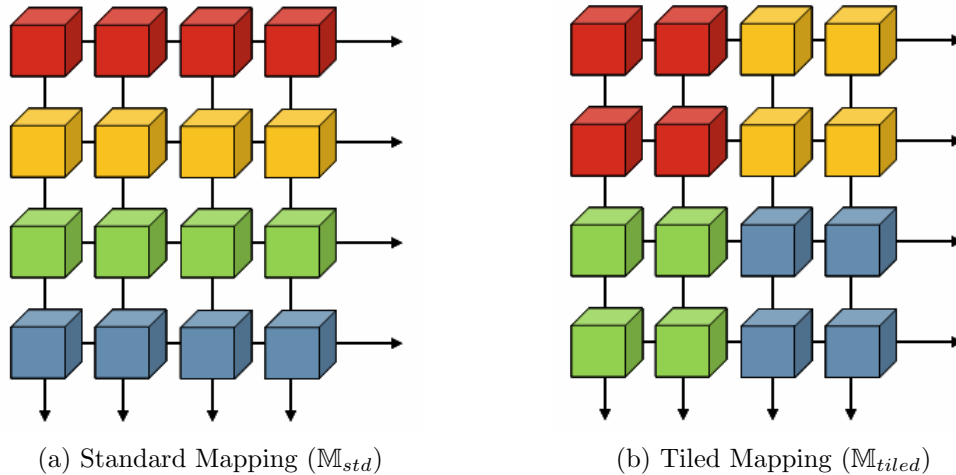


Figure 5.6: Example of a 2D mappings with $n = 4$ and $k = 4$.

$$f_{i,1}(\mathbb{N}_{std}) = \{0, 0, 0, 0\}$$

$$f_{i,2}(\mathbb{N}_{std}) = \{4, 4, 4, 0\}$$

$$f_{i,1}(\mathbb{N}_{tiled}) = \{2, 0, 2, 0\}$$

$$f_{i,2}(\mathbb{N}_{tiled}) = \{2, 2, 0, 0\}$$

5.3 Theoretical Analysis

We will analyze the Multi-dimensional MPI Communication based on two different algorithms; the Pipeline algorithm and the Binomial Tree algorithm. Before we can do that we need to define some variables which help us to describe the *multidimensional communication*.

Let:

- d be the number of dimensions
- m the size of the message
- s size of a message segment
- k_i the number of cores per node in the i^{th} dimension

such that $k = \prod_{i=1}^d k_i$

- n_i the number of nodes in the i^{th} dimension
such that $n = \prod_{i=1}^d n_i$
- p_i the number of processes in the i^{th} dimension
such that $p = \prod_{i=1}^d p_i$

5.3.1 Multi-Dimensional MPI Communication Based on Pipeline Algorithm

Since many of the current applications perform their computations with data that is sent between the computational processes by means of collective communications; it is evident that optimizing the multidimensional grids and how they are employed is paramount to the overall effectiveness of the processes. To show how different algorithms create different maps, we will begin by using the pipeline communication as an example. MPICH2 and OpenMPI are two major implementations of the MPI standard. Both methods divide large messages into smaller fragments in order to establish a pipeline [68, 69]. Our assumption is that the communication typically uses larger message sizes. Therefore, we implement a chain-pipeline encoding algorithm as described by Chen and Dongarra [70]:

In p -processor system, where $p = n \cdot k$ and n denotes the number of nodes and k the amount of cores per node, first, organize everything as a chain (Figure 5.7). Second, divide the data on each process into many small pieces, δ segments all of size s such that $m = \delta \cdot s$. The j^{th} segment of $m[i]$ is denoted as $m[i][j]$. Third, calculate:

$$\sum_{i=0}^{p-1} m[i]$$

in a pipeline fashion by calculating

$$\sum_{i=0}^{p-1} m[i] = \sum_{i=0}^{p-1} \sum_{j=0}^{\delta-1} m[i][j]$$

The chain-pipelined encoding can be modeled by $T_{step} = \alpha + \beta \cdot s + \gamma \cdot s$. There are $n \cdot k$ processors in our communication group and therefore $n \cdot k - 1$ steps before the last process receives the first segment. There are $\delta - 1$ segments left to deliver after the pipeline

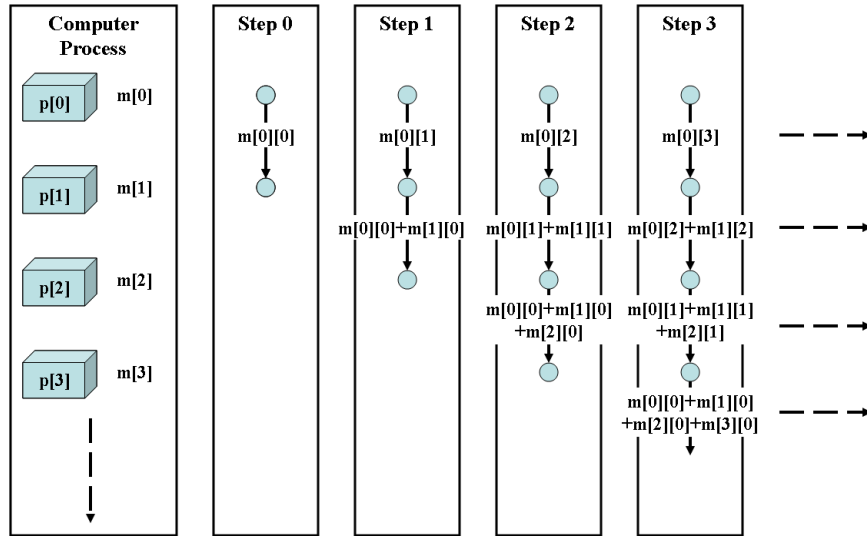


Figure 5.7: Reduction function using the Pipeline Algorithm

is established, and therefore $\delta - 1$ steps before the last process receives the last segment. The number of steps to encode and deliver δ segments in a $n \cdot k$ processor communication group is $(n \cdot k - 1) + (\delta - 1)$, thus the total time becomes:

$$\begin{aligned}
 T_{tot}(m) &= [(n \cdot k - 1) + (\delta - 1)] (\alpha + \beta \cdot s + \gamma s) \\
 &= \left(n \cdot k - 2 + \frac{m}{s} \right) (\alpha + \beta \cdot s + \gamma s)
 \end{aligned} \tag{5.2}$$

5.3.2 Two-Dimensional Case

The obvious solution in a one dimensional pipeline is to string the nodes one after another. However, it does tell us that that `MPI_Init()` and `MPI_Cart_create()` in the cases of `Round-robin` and `Folded rank` create a non-optimal solution.

Many of today's applications demand the programmer to arrange the processes in a matrix and perform collective reduction communication over both the rows and the columns (Figure 5.8).

Let the $n \cdot k$ processes be arrange in a $p_1 \times p_2$ matrix such that $p = n \cdot k = p_1 \cdot p_2$, and $k|p$. Further, let each process have a message of size $m = \delta \cdot s$, where δ is the the amount of segments of size s the message is divided into. Let this message be pipelined both along the

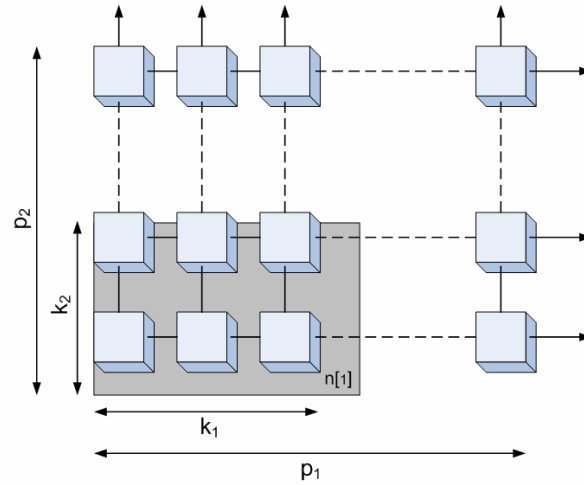


Figure 5.8: Matrix reduction in 2 dimensions

rows and the columns, and let the total time be:

$$T_{tot} = \max [T(\text{row}_i)] + \max [T(\text{column}_i)]$$

There are p_1 processes in each row. Each row will therefore complete $q_1 - 1$ steps before the last process of that row has received the first segment of the message and there are $\delta - 1$ segments left to deliver. Assuming that $\alpha_E = \alpha_I$ and the time to perform the calculation on s is equal to $\gamma \cdot s$ on all processes. We can use Equation 5.2 to estimate the total time for the pipeline over each row for some mapping \mathbb{M}_j as:

$$T_{row}(\mathbb{M}_j) = \left(p_1 - 2 + \frac{m}{s} \right) (\alpha + \gamma s) + T_{send}$$

Each column will complete $p_2 - 1$ steps before the last process of that column has received the first segment of the message and there are $\delta - 1$ segments left to deliver. Making the same assumption regarding α and γ as in the rows. The total time for the pipeline over each column for some mapping \mathbb{M}_j using Equation 5.2 is estimated to:

$$T_{column}(\mathbb{M}_j) = \left(p_2 - 2 + \frac{m}{s} \right) (\alpha + \gamma s) + T_{send}$$

by adding two results together we get:

$$\begin{aligned}
T_{tot}(\mathbb{M}_j) &= \left(p_1 - 2 + \frac{m}{s}\right) (\alpha + \gamma s) + T_{send} \\
&+ \left(p_2 - 2 + \frac{m}{s}\right) (\alpha + \gamma s) + T_{send} \\
&= \left((p_1 + p_2) - 4 + \frac{2m}{s}\right) (\alpha + \gamma s) + T_{send}
\end{aligned} \tag{5.3}$$

Let:

$$\begin{aligned}
E &= \max [f_{0,1}(\mathbb{N}_j), f_{1,1}(\mathbb{N}_j), \dots, f_{n-1,1}(\mathbb{N}_j)] \\
&+ \max [f_{0,2}(\mathbb{N}_j), f_{1,2}(\mathbb{N}_j), \dots, f_{n-1,2}(\mathbb{N}_j)]
\end{aligned}$$

Where $f_{i,d}(\mathbb{N}_j)$ is defined by Definition 5.1. It is important to notice that the length of any single pipeline is *only* the number of rows or columns (Figure 5.6). The communication will be parallel and highly dependent on the number of channels available for external communication. We can therefore set T_{send} equal to:

$$\begin{aligned}
T_{send}(\mathbb{M}_j) &= \left((p_1 + p_2) - 4 + \frac{2m}{s}\right) \\
&\left[\frac{E}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} + \frac{(p_1 + p_2) - E}{(p_1 + p_2)} \cdot \beta_I \right] s
\end{aligned}$$

using this result in Equation 5.3 results in the following total time:

$$\begin{aligned}
T_{tot}(\mathbb{M}_j) &= \left((p_1 + p_2) - 4 + \frac{2m}{s}\right) \left(\alpha + \gamma s + \right. \\
&\left. s \left[\frac{E}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} + \frac{(p_1 + p_2) - E}{(p_1 + p_2)} \cdot \beta_I \right] \right)
\end{aligned}$$

Our optimal solution can be defined by the same optimization function (Equation 4.2) as for the single node case:

$$\min [T(\mathbb{M}_0), T(\mathbb{M}_1), \dots, T(\mathbb{M}_{\|\mathbb{M}\|})]$$

If we claim that $\beta_E \gg \beta_I$, we can conjecture that this solution will be found in a mapping \mathbb{M}_j s.t. we have E_{min} . In other words, to conduct a matrix-reduction as fast as possible we need to create a communicator such that the number of **External** communication links

from each node is equal to or less than the amount of channels available.

A node with 8 cores could for example be arranged as a 1×8 tile, 2×4 tile, 4×2 tile or 8×1 tile, with the numbers of **External** links equal to 9, 6, 6 and 9. In this case it is easy to conjecture that either a 2×4 or 4×2 tiling would provide an optimal solution. In a node with 12 cores we have even more options, tiles of size: 1×12 , 2×6 , 3×4 , 4×3 , 6×2 , 12×1 are possible, with the numbers of **External** links equal to 13, 8, 7, 7, 8 and 12.

It is relatively easy to find an optimal solution in these small cases but what happens when the number of cores k per node reaches 128 or above? What if the size of our matrix is in the range of 1000s of rows by 1000s of columns? We need a way to automatically calculate an optimal solution.

Let:

A denote the set of all positive divisors of p_1

B denote the set of all positive divisors of k

$$q_i = \begin{cases} b_i + \frac{k}{b_i} & \text{if } b_i \in A \text{ and } \frac{k}{b_i} | p_2 \\ \infty & \text{else} \end{cases}$$

Find the $\min[q_0, q_1, \dots, q_{||\mathbf{q}||}]$ and let $k_1 = b_i$ and $k_2 = \frac{k}{b_i}$ for the b_i associated with this \min .

This algorithm will find a solution. Depending on the process-grid layout it might not be an optimal solution, but as it is required that $k|p$, the algorithm will at least find the trivial solutions ($k_1 = 1$, $k_2 = k$ or $k_1 = k$, $k_2 = 1$).

Now create a **rank**-vector such that each node n_i is arranged as $k_1 \times k_2$ tile. Given this **rank**-vector and Definition 5.1 we know that:

$$\begin{aligned} E &= \max[f_{0,1}(\mathbb{N}_j), f_{1,1}(\mathbb{N}_j), \dots, f_{n-1,1}(\mathbb{N}_j)] \\ &+ \max[f_{0,2}(\mathbb{N}_j), f_{0,2}(\mathbb{N}_j), \dots, f_{n-1,2}(\mathbb{N}_j)] \\ &= k_1 + k_2 \end{aligned}$$

using the fact that $E \leq c$ our optimal solution becomes:

Algorithm 5.11: Automatically find a 2D tile layout

Input: The number of cores in a node k and number of processes in each dimension

p_1, p_2

Output: Number of cores in each dimension

```
1 B ← all positive divisors of  $k$ 
2  $min \leftarrow \infty$ 
3 for  $i \leftarrow 0$  to number of elements in B do
4   | if  $b_i | p_1$  and  $\frac{k}{b_i} | p_2$  then
5   |   |  $sum \leftarrow b_i + \frac{k}{b_i}$ 
6   |   | if  $sum < min$  then
7   |   |   |  $k_1 \leftarrow b_i$ 
8   |   |   |  $min \leftarrow sum$ 
9   |   | end
10  | end
11 end
12  $k_2 \leftarrow \frac{k}{k_1}$ 
13 Display  $k_1$  and  $k_2$ 
```

Algorithm 5.12: Create a tiling of a 2D process-grid

Input: The dimensions of the core k_1, k_2 , number of processes in each dimension

p_1, p_2 and a SMP-ordered rank vector **rank**

Output: A tiled rank vector **rank**

```
1  $k \leftarrow k_1 \cdot k_2$ 
2 for  $r \leftarrow 0$  to  $p_2$  do
3   | for  $c \leftarrow 0$  to  $p_1$  do
4   |   | if  $k_1 | c$  then
5   |   |   |  $i \leftarrow (r \% k_2) \cdot k_1 + \lfloor r/k_2 \rfloor \cdot (p_1 \cdot k_2) + \lfloor c/k_1 \rfloor \cdot k$ 
6   |   |   | end
7   |   |   | else
8   |   |   |   |  $i \leftarrow i + 1$ 
9   |   |   |   | end
10  |   |   | rank $_{r \cdot p_1 + c} \leftarrow i$ 
11  |   | end
12 end
13 Display rank
```

$$\begin{aligned}
T_{tot}(\mathbb{M}_j) &= \left((p_1 + p_2) - 4 + \frac{m}{s} \right) \left(\alpha + \gamma s + \right. \\
&\quad \left. s \left[\frac{k_1 + k_2}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} + \frac{(p_1 + p_2) - (k_1 + k_2)}{(p_1 + p_2)} \cdot \beta_I \right] \right) \\
&= \left((p_1 + p_2) - 4 + \frac{2m}{s} \right) \left(\alpha + \gamma s + \right. \\
&\quad \left. s \left[\frac{\beta_E}{p_1 + p_2} + \frac{(p_1 + p_2) - c}{(p_1 + p_2)} \cdot \beta_I \right] \right) \tag{5.4}
\end{aligned}$$

We can use this result to make a rough estimate regarding performance improvement as the number of cores per node increases. Assuming the time for communication will be dominant over the setup and computational times, also assume that $\beta_E \gg \beta_I$ ($\beta_I = 0$) then:

$$\begin{aligned}
T_{tot}(\mathbb{M}_j) &= \left((p_1 + p_2) - 4 + \frac{2m}{s} \right) s \left[\frac{k_1 + k_2}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} \right] \\
\Theta &= \frac{\left((p_1 + p_2) - 4 + \frac{2m}{s} \right) \left(s \left[\frac{k_1 + k_2}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} \right] \right)}{\left((p_1 + p_2) - 4 + \frac{2m}{s} \right) \left(s \left[\frac{k'_1 + k'_2}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} \right] \right)} \tag{5.5} \\
&= c \cdot \frac{k_1 + k_2}{k'_1 + k'_2}
\end{aligned}$$

What values can $k_1 + k_2$ take? We know improvement is related to the ratio between the largest and smallest value for the periphery where the area is given by the number of cores k on the node. The largest possible value is therefore $k_1 + k_2 = k + 1$ and the smallest possible would be a perfect square $k'_1 + k'_2 = \sqrt{k} + \sqrt{k}$. Our upper bound becomes:

$$\Theta = c \cdot \frac{k + 1}{2 \cdot \sqrt{k}} \approx \frac{1}{2} \cdot c \cdot \sqrt{k}$$

5.3.3 d -Dimensional Matrix Arrays

We extend it to a d -dimensional matrix, by letting the $n \cdot k$ processes be arranged in a $p_1 \times p_2 \dots \times p_d$ matrix such that $p = n \cdot k = p_1 \cdot p_2 \dots \cdot p_d$. Further, let each process have a message of size $m = \delta \cdot s$, where δ such message is simultaneously pipelined in every dimension the total time can be estimated to the sum of the slowest throughput in each dimension:

$$T_{tot} = \max [T(d[1]_i)] + \max [T(d[2]_i)] \dots \max [T(d[d]_i)]$$

There are p_i ($i = 1 \dots d$) processes in each dimension. Each dimension will therefore complete $p_i - 1$ steps before the last process of that dimension has received the first segment of the message and there are $\delta - 1$ segments left to deliver. The estimated total time for the pipeline over each dimension will equal (Equation 5.2):

$$T_i(\mathbb{M}_j) = \left(p_i - 2 + \frac{m}{s} \right) (\alpha + \gamma s) + T_{send}$$

adding the d -dimensions together yields:

$$\begin{aligned} T_{tot}(\mathbb{M}_j) &= \sum_{i=1}^d \left(p_i - 2 + \frac{m}{s} \right) (\alpha + \gamma s) + T_{send} \\ &= \left(\sum_{i=1}^d p_i - 6 + \frac{3m}{s} \right) (\alpha + \gamma s) + T_{send} \end{aligned} \quad (5.6)$$

As in the $2D$ -case the length of any single pipeline is equal in length to any single row in each dimension (see Equation 5.4). Looking at a single dimension (j) we can see that there is a total of p_j communicators of length $\prod_{i \neq j}^d p_i$, and the T_{send} is therefore equal to:

$$\begin{aligned} T_{send}(\mathbb{M}_j) &= \left(\sum_{i=1}^d p_i - 2d + \frac{dm}{s} \right) \\ &\quad s \left[\frac{E}{\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d p_j} \cdot \frac{\beta_E}{c} + \right. \\ &\quad \left. \frac{\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d k_j - E}{\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d p_j} \cdot \beta_I \right] \end{aligned} \quad (5.7)$$

where c denotes the number of messages that can be sent in parallel without reduction in performance, but what is E ? E describes just as in the $2D$ -case the amount of *External* messages over all dimensions. Using Definition 5.1 we can define E as:

$$\begin{aligned} E &= \max [f_{0,1}(\mathbb{N}_j), f_{1,1}(\mathbb{N}_j), \dots, f_{n-1,1}(\mathbb{N}_j)] + \dots \\ &\quad + \max [f_{0,d}(\mathbb{N}_j), f_{0,d}(\mathbb{N}_j), \dots, f_{n-1,d}(\mathbb{N}_j)] \end{aligned} \quad (5.8)$$

Applying the three Equations 5.6, 5.7 and 5.8 total time over the d -dimensions becomes:

$$\begin{aligned}
T_{tot}(\mathbb{M}_j) &= \left(\sum_{i=1}^d p_i - 2d + \frac{dm}{s} \right) \left(\alpha + \gamma\sigma \right. \\
&+ \sigma \left[\frac{E}{\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d p_j} \cdot \frac{\beta_E}{c} \right. \\
&\left. \left. + \frac{\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d p_j - E}{\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d p_j} \cdot \beta_I \right] \right)
\end{aligned}$$

The optimal solution will be found by minimizing the ratio between the d -dimensional *hypervolume* of the nodes

$$\prod_{i=1}^d k_i$$

and the d -dimensional *hyper-surface-area* of the nodes

$$\sum_{i=1}^d \prod_{\substack{j=1 \\ j \neq i}}^d k_j$$

Using Equation 5.5 we conclude that the communication improvement from an optimally tiled mapping over the default mapping is on the order of:

$$\Theta \left(c \cdot \frac{(d-1)k+1}{d \cdot (k^{1/d})^{d-1}} \right) = \Theta \left(\frac{d-1}{d} \cdot c \cdot k^{1/d} \right)$$

It should be noted that when $k^{1/d} < 2$ we achieve no further improvements aside from optimizing the tiling for $(d-1)$ -dimensional communication.

5.3.4 Multi-Dimensional MPI Communications Based on Binomial Tree Algorithm

Pipelined communication is just one method among many MPI communicators. Many of the others are built upon the binomial tree (Figure 5.9), wherein the default **rank** used with any collective communicator in dimensions higher than one would use only *External* communication in every dimension but one. We also know from the structure of the binomial tree communicator that the first step in gather (or last step in scatter), represents half of

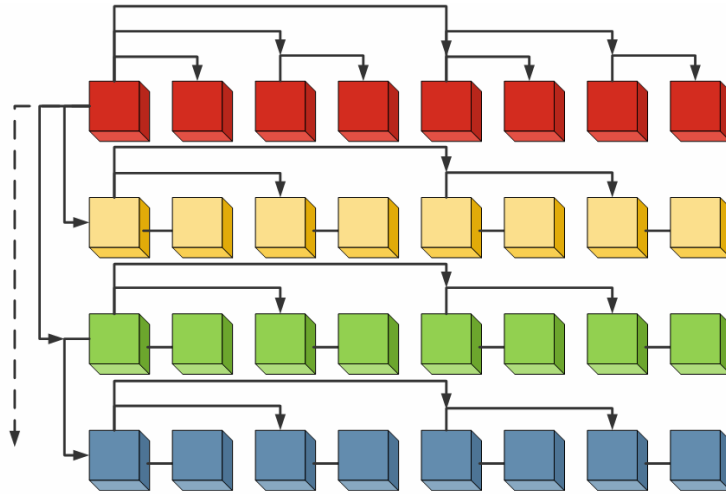


Figure 5.9: Example of a 2D binomial tree communication grid, each process belongs to both a row- and column-communicator

the point-to-point communication (Figure 5.10).

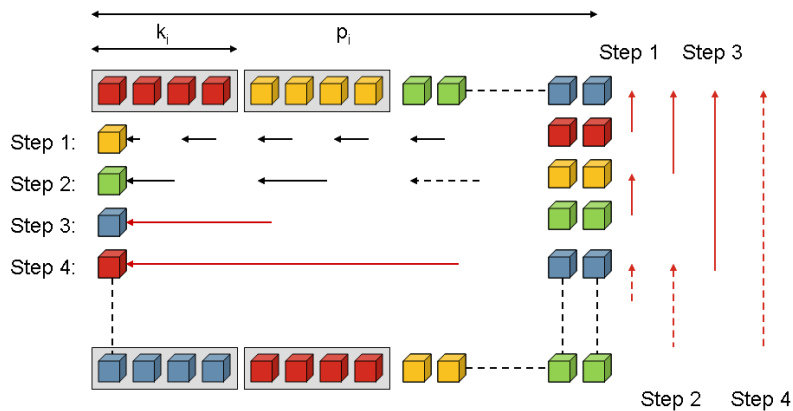


Figure 5.10: Example of a non-tiled 2D binomial tree communication grid. Black arrows show Internal communication and red arrows External communication

We would expect major improvements if it was possible to ensure that this first (or last step) was *Internal* communication on all nodes. We would then cut the external communication by as much as 50% (Figure 5.11).

Admittedly, by re-arranging the nodes so they have at least two cores in each dimension, we would most likely increase the amount of *External* communication in the other dimensions, but we would still achieve an improvement if the total increase is smaller than 50%.

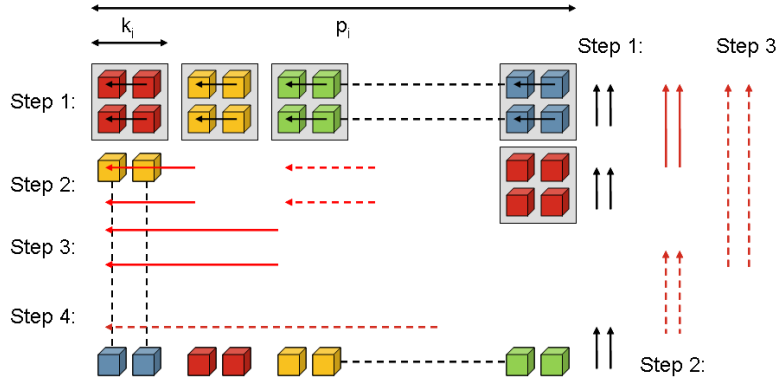


Figure 5.11: Example of a tiled 2D binomial tree communication grid. Black arrows show Internal communication and red arrows External communication

To analyze this problem we first define k_i as $k_i = q_i \cdot 2^{r_i}$, where r_i is an integer and q_i an odd integer. By definition will k equal:

$$k = \prod_{i=1}^d k_i = \prod_{i=1}^d q_i \cdot 2^{r_i}$$

The communication time for the binomial tree over a homogeneous network is defined as:

$$T = (\alpha + \beta \cdot m) \lceil \log_2 p \rceil$$

In a multicore architecture we distinguish between *Internal* and *External* communication. The time through a single dimensional communication will therefore be defined by:

$$T = (\alpha + \beta_E \cdot m) \cdot (\lceil \log_2 n \cdot k \rceil - r) + (\alpha + \beta_I \cdot m) \cdot r$$

where $k = q \cdot 2^r$ is the number of cores in each of the n nodes.

When we extend this to d -dimension, we will assume that all communication in each dimension is in parallel. We therefore have to adjust the equation to reflect not only the size of the node in i^{th} direction, but also account for the number of channels available. Let the communication time in the i^{th} dimension T_i be defined as:

$$\left\lceil \frac{k_i}{A_i} \cdot \frac{1}{c} \right\rceil \left\{ (\alpha + \beta \cdot m) \cdot (\lceil \log_2 p_i \rceil - r_i) + (\alpha + \beta_I \cdot m) \cdot r_i \right\}$$

Where p_i is number of processes in the i^{th} dimension, k the number of cores and A_i is the *hyper-surface-area* in the i^{th} -dimension of the body created by the k cores.

The total time is defined as the summation over all dimensions.

$$T = \sum_{i=1}^d T_i = \sum_{i=1}^d \left[\frac{k_i}{\prod_{\substack{j=1 \\ j \neq i}}^d k_j} \cdot \frac{1}{c} \right] \left\{ (\alpha + \beta \cdot m) \cdot (\lceil \log_2 p_i \rceil - r_i) + (\alpha + \beta_I \cdot m) \cdot r_i \right\}$$

The objective function $\min_{k_i, r_i} \{T\}$, to minimize T through k_i and r_i

5.4 Evaluation

A series of tests were conducted to perform the evaluation. The first test was the *Extended Parallel Ping-Pong Test* (See Chapter 3), designed to evaluate the difference between *Internal* and *External* throughput. The main tests were conducted after that on multi-dimensional grids. All these tests compared the default MPI `communicator`, using *SMP-style* mapping, against an optimized `Tiled communicator`, using the following communication patterns:

- 3D Broadcast (based on pipeline)
- 2D Broadcast (based on pipeline)
- 2D Broadcast (based on binomial tree)
- 2D Gather
- 2D Scatter

Three different systems were used for these experiment; *Alamode*, *RA* and *Jaguar* (see Chapter 3.7.1 for specifications). Our purpose is not to compare the systems against each other, but to compare the implementations of tiled communicator versus the standard communicator. The message size m is therefore not kept constant between the different systems, but have been chosen depending on the bandwidth $1/\beta$ and the latency α for each system.

5.4.1 *Alamode* Results

On this system we expected a significant part of the performance improvement to be contributed from the fact that the difference between *Internal* and *External* communication have a magnitude close to a factor of 10. The standard **rank** created by `MPI_Init()` and `MPI_Cart_create()` is **SMP**-style.

Assume that the majority of the total time can be contributed to the sending, and not the setup or calculation, $\beta_E \gg \beta_I$ and Θ defined by Equation 5.5 we can then derive our performance improvement as:

$$\begin{aligned} \Theta &= \frac{\left((p_1 + p_2) - 4 + \frac{2m}{s} \right) \left(s \left[\frac{k_1 + k_2}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} \right] \right)}{\left((p_1 + p_2) - 4 + \frac{2m}{s} \right) \left(s \left[\frac{k'_1 + k'_2}{(p_1 + p_2)} \cdot \frac{\beta_E}{c} \right] \right)} \\ &= \frac{k_1 + k_2}{k'_1 + k'_2} = \frac{1 + 4}{2 + 2} = 1.2 \end{aligned}$$

The improvement for the functions built on the binomial trees (`MPI_Sctr()` and Binomial Broadcast) achieve more than 30% improvement which is expected (see Figure 5.12 and Figure 5.14(c)).

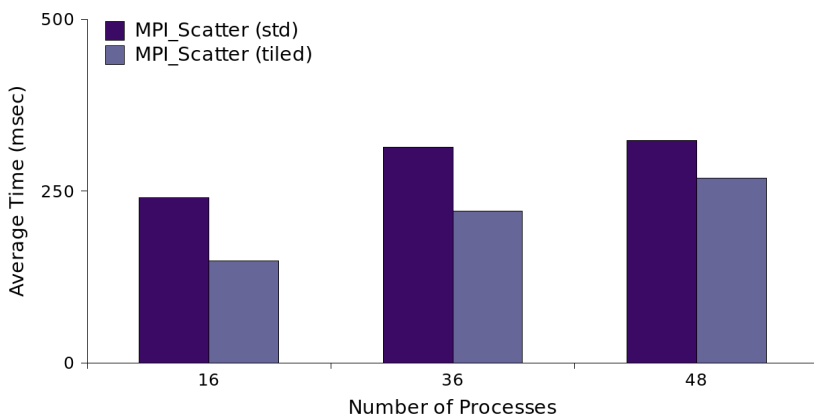


Figure 5.12: Two Dimensional `MPI_Sctr()` on the *Alamode* System($m = 8 MB$).

The `MPI_Gthr()`, show less of improvement than the scatter function, but the improvement is still a 25% and within the expected values (see Figure 5.13).

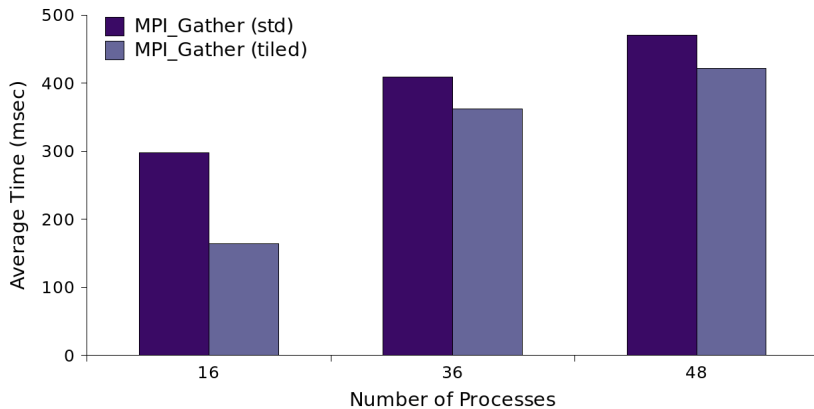


Figure 5.13: Two Dimensional MPI_Gthr() on the *Alamode* System($m = 8 MB$).

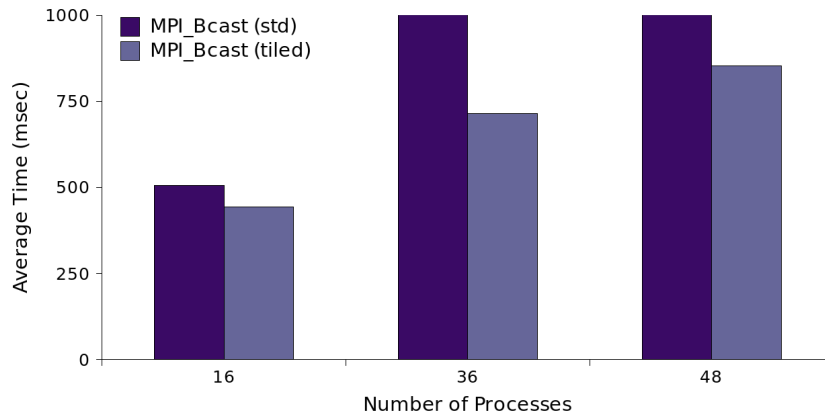
Finally the *broadcast* tests show the expected improvement for all three types (see Figure 5.14). We do have a slightly better improvement when using `MPI_Bcast()` than we would have expected (see Figure 5.14(a)). This is consistent over all tests, it is also on the on the standard MPI broadcast function, we will therefore attribute this slightly higher than expected improvement to dynamic adaptations in the MPI function itself.

5.4.2 RA Results

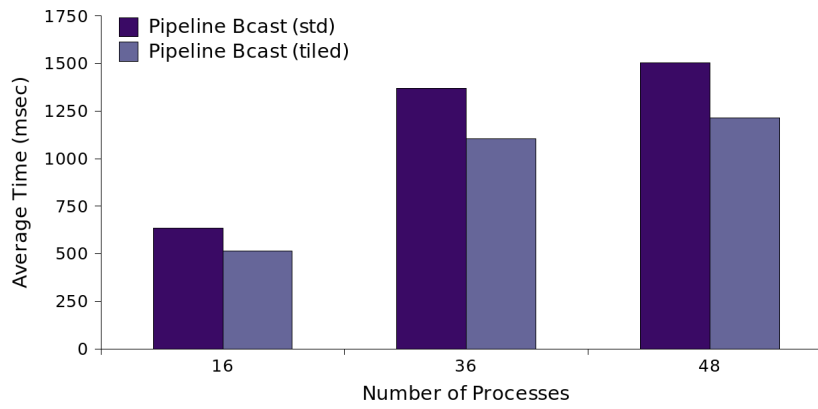
In the two dimensional communication the system has more cores per node and should therefore allow for *tiles* that grant more *Internal* communication in the two dimensional grid. Assuming perfect *Internal* communication ($\beta_I = 0$) and Θ defined by Equation 5.5 we can make a rough estimate regarding performance increase:

$$\Theta = \frac{k_1 + k_2}{k'_1 + k'_2} = \frac{1 + 8}{2 + 4} = 1.5$$

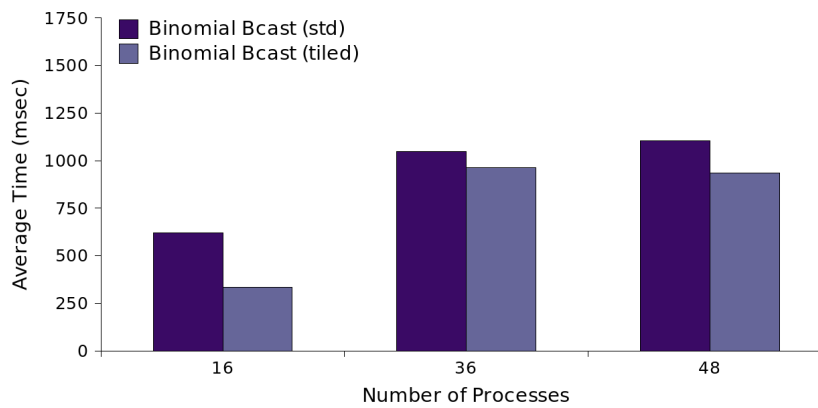
This is the upper bound of what we can expect from *re-mapping* 8 processors. We know that the difference between *Internal* and *External* communication are not as significant and thus, we should not expect to see values close to this limit (Figure 5.15, Figure 5.16 and Figure 5.17). The standard **rank** created by `MPI_Init()` and `MPI_Cart_create()` is **SMP**-style.



(a) Two Dimensional MPI_Bcast()

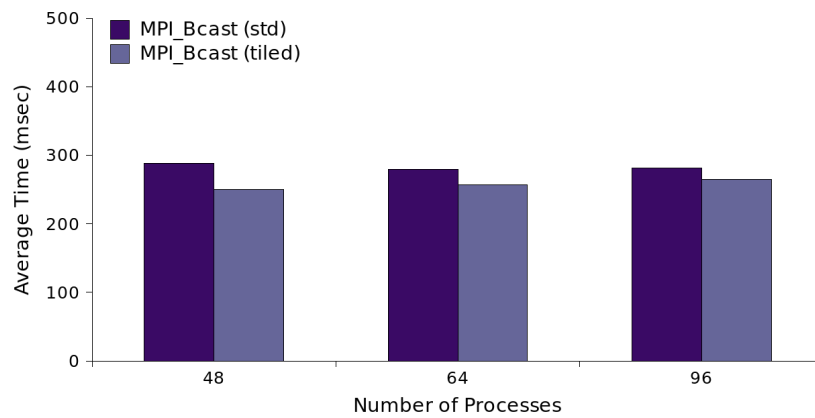


(b) Two Dimensional Broadcast Using Pipeline Algorithm

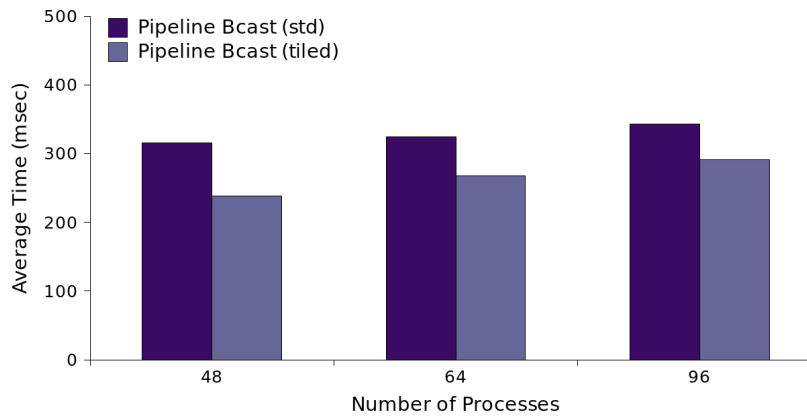


(c) Two Dimensional Broadcast Using Binomial Tree Algorithm

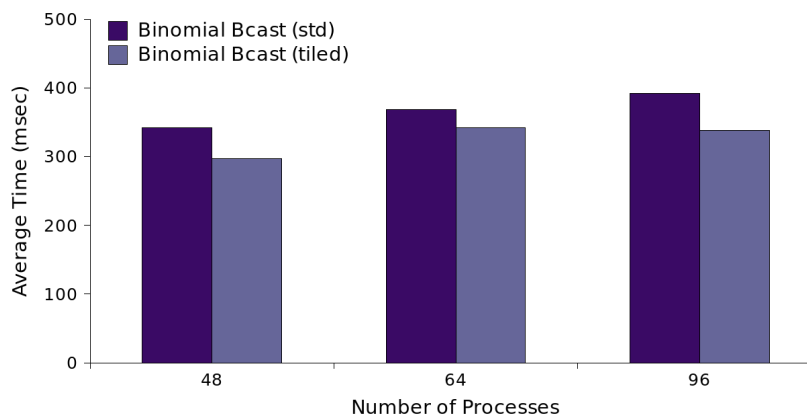
Figure 5.14: Broadcast Functions on the Alamode System ($m = 8 MB$).



(a) Two Dimensional MPI_Bcast()



(b) Two Dimensional Broadcast Using Pipeline Algorithm



(c) Two Dimensional Broadcast Using Binomial Tree Algorithm

Figure 5.15: Broadcast Functions on the RA Cluster ($m = 16 MB$).

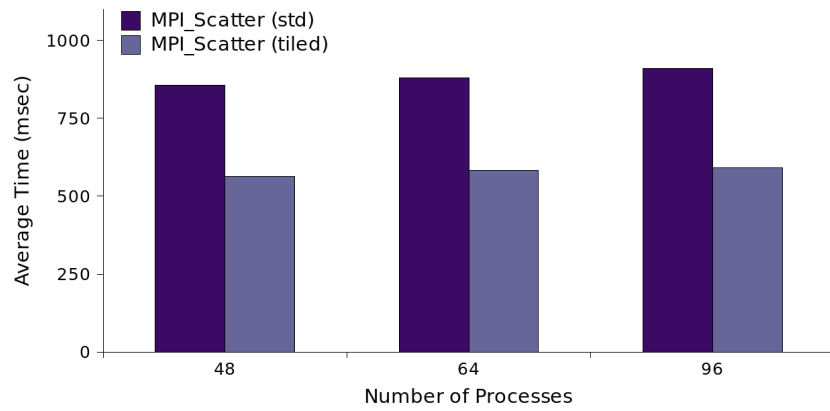


Figure 5.16: Two Dimensional MPI_Scatter() on the RA Cluster ($m = 16 MB$).

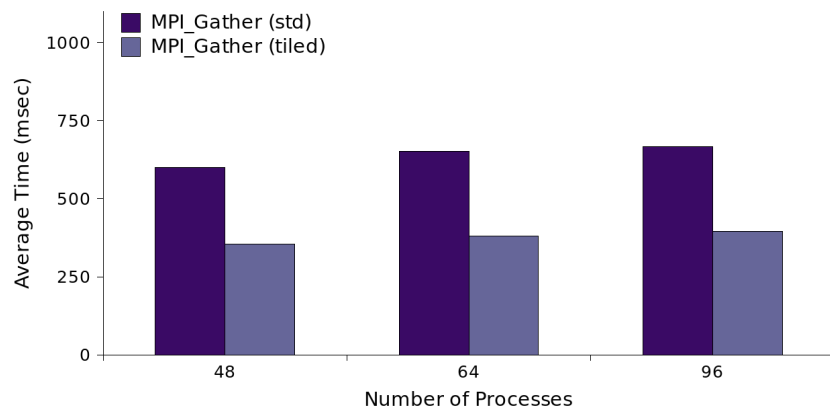


Figure 5.17: Two Dimensional MPI_Gather() on the RA Cluster ($m = 16 MB$).

A more accurate estimate of the improvement would be to assume that the majority of the total time can be contributed to the sending and not setup or calculation. Letting $\beta_E \approx \frac{1271 \cdot \beta_I}{1021}$ we can get a rough but more accurate estimate of the improvement:

$$\begin{aligned}
\Theta &\approx \frac{\left(s \left[\frac{k_1+k_2}{(p_1+p_2)} \cdot \frac{\beta_I \cdot 1280}{c \cdot 1020} + \frac{(p_1+p_2)-(k_1+k_2)}{(p_1+p_2)} \cdot \beta_I \right] \right)}{\left(s \left[\frac{k'_1+k'_2}{(p_1+p_2)} \cdot \frac{\beta_I \cdot 1280}{c \cdot 1020} + \frac{(p_1+p_2)-(k'_1+k'_2)}{(p_1+p_2)} \cdot \beta_I \right] \right)} \\
&\approx \frac{(k_1 + k_2) + \frac{32}{51}((p_1 + p_2) - (k_1 + k_2))}{(k'_1 + k'_2) + \frac{32}{51}((p_1 + p_2) - (k'_1 + k'_2))} \\
&\approx \frac{(1 + 8) + .63 \cdot ((14 + 16 + 20)/3 - 9)}{(2 + 4) + .63 \cdot (17 - 6)} = 1.18
\end{aligned}$$

This low estimate is very close to our experimental results. The improvement based on the broadcast functions are about 10-15% and for the functions with underlying binomial trees the improvement is more than 50%.

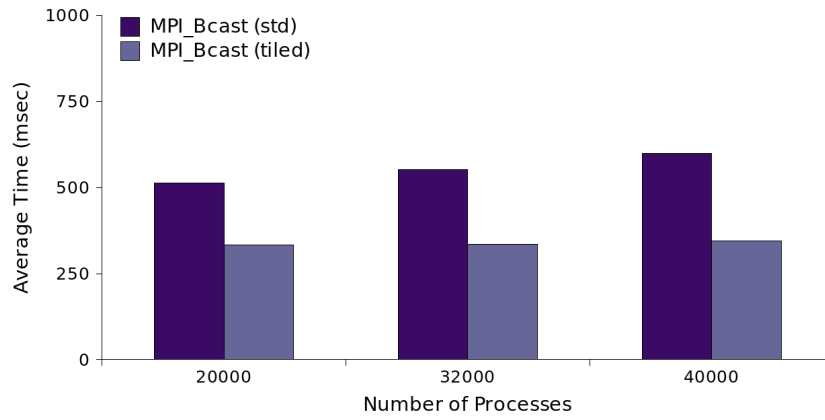
5.4.3 Jaguar Results

For the two dimensional case we once again estimate the upper bound for the performance improvement to:

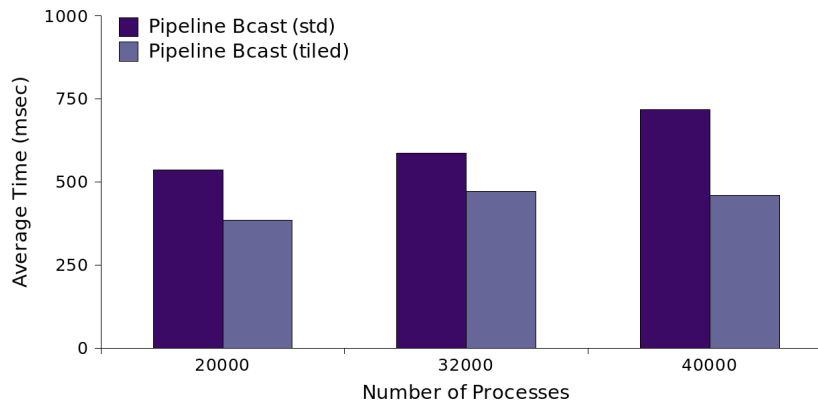
$$\Theta = \frac{k_1 + k_2}{k'_1 + k'_2} = \frac{1 + 12}{3 + 4} = 1.86$$

Once again we let the standard **rank** created by `MPI_Init()` and `MPI_Cart_create()` be **SMP**-style. This time a significant part of the performance improvement should be contributed to the number of cores per node. We are allowed to create *tiles* that have almost an optimal ratio between area and perimeter (Figure 5.18, Figure 5.19 and Figure 5.20).

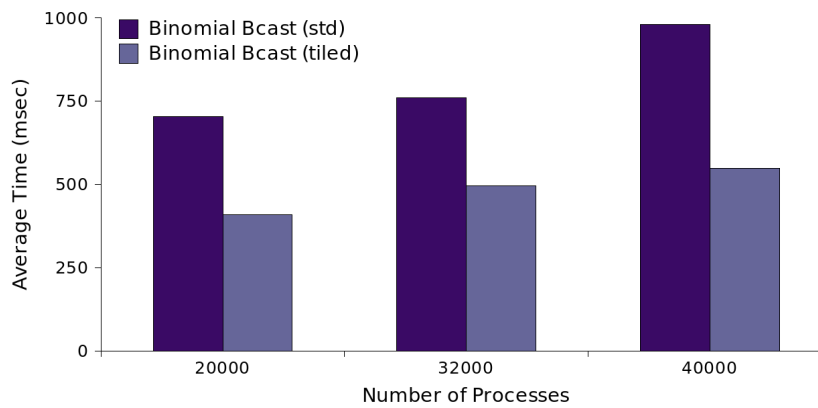
There is a noticeable change in *throughput* between $p = 10,000$ and $p = 20,000$ cores (see Figure 5.21). This change is attributed to switch level topologies; the request changes from a *large* to a *very large* job and it has to be queued much longer before executing, and most likely been assigned resources with a physical locality close to each other. It should also be noticed that `MPI_Bcast()` do not scale linearly. The experiments confirms $\beta_E \gg \beta_I$ shown by the fact the performance improvement is close to the estimated *upper bound*.



(a) Two Dimensional MPI_Bcast()



(b) Two Dimensional Broadcast Using Pipeline Algorithm



(c) Two Dimensional Broadcast Using Binomial Tree Algorithm

Figure 5.18: *Broadcast* Functions on the *Jaguar* Cluster ($m = 64$ MB).

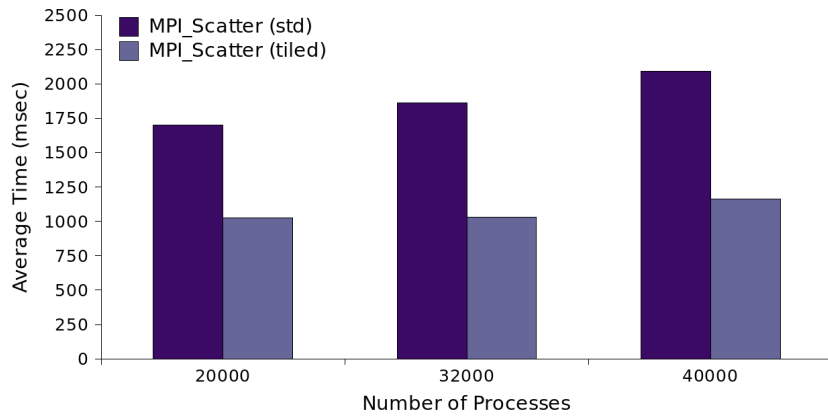


Figure 5.19: Two Dimensional MPI_Sctr() on the *Jaguar* Cluster ($m = 64 MB$).

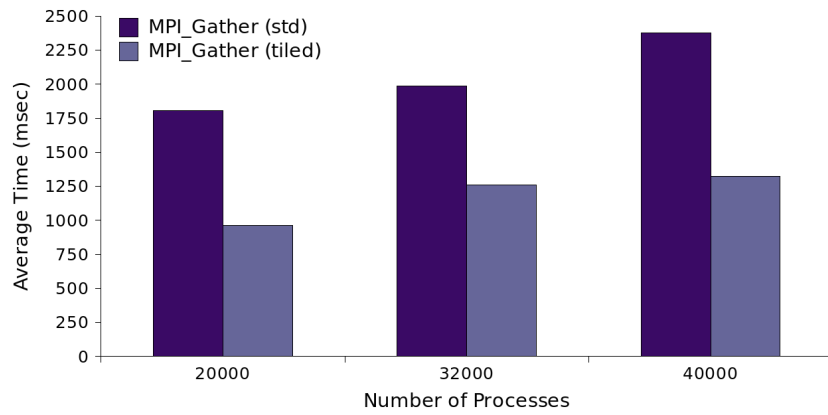


Figure 5.20: Two Dimensional MPI_Gthr() on the *Jaguar* Cluster ($m = 64 MB$).

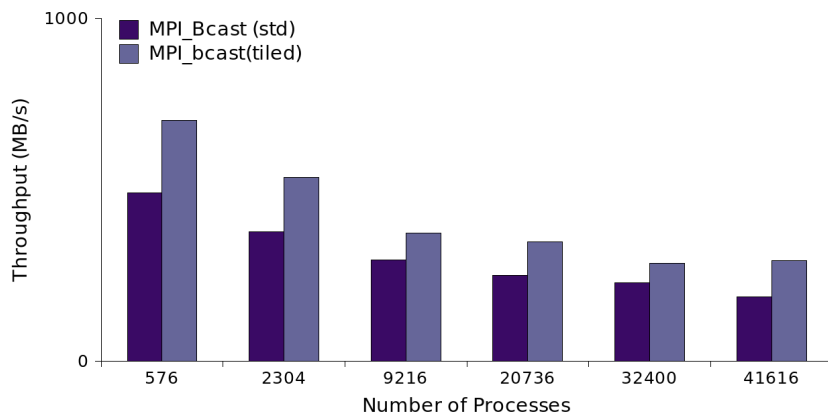


Figure 5.21: Throughput values for MPI_Bcst() on the *Jaguar* Cluster ($m = 64 MB$).

5.4.4 Combined Tables for the Two-Dimensional Results on all Systems

Table 5.2, Table 5.3 and Table 5.4 show the combined results for all the two-dimensional tests. The combined results verify that there is significant improvement to tile the nodes when using many of the collective communication models.

5.4.5 Three-Dimensional Results on *RA* and *Jaguar*

The 3D tests used a **Broadcast** function based on pipeline and were only conducted on *RA* and *Jaguar* (see Table 5.1). We excluded *Alamode* because the four cores do not allow any re-configuration in a three dimensional tile. There was only one efficient way to tile the cores on *RA* when using the 3D model, a $(2 \times 2 \times 2)$ block. For *Jaguar* the only efficient tile was a $2 \times 2 \times 3$ block configuration. This means that the upper-bound estimate for *RA* would be:

$$\Theta = \frac{2 \cdot k + 1}{BA} = \frac{2 \cdot 8 + 1}{4 + 4 + 4} = 1.42$$

while the same bound on *Jaguar* would be be:

$$\Theta = \frac{2 \cdot k + 1}{BA} = \frac{2 \cdot 12 + 1}{4 + 6 + 6} = 1.56$$

The average improvement on *RA* was only 10% which can most likely be contributed to the relatively small grids and the fact that both the *External* and *Internal* bandwidths were very similar. Yet, the average improvement on *Jaguar* was 33%. As opposed to *RA*, there was a significant difference in the *External* and *Internal* bandwidths of *Jaguar* and the size of the grids were also much larger.

5.4.6 Re-mapping Overhead

We have shown that the *tiling* technique improves the performance of many of the collective communication functions, however, the *tiling* technique itself introduces overhead. The overhead for creating the tiled process-to-core mapping will be evaluated in this subsection.

Table 5.1: MPI_Bcast() Tests on *RA* and *Jaguar* Implementing Three-Dimensional Communication Grids (16 MB message size).

Grid Size	Communicator			
	Standard		Tiled	
<i>RA</i>	Time (msec)	stddev	Time (msec)	stddev
$p = 48$	705.29	10.91	653.04	9.76
$p = 64$	1160.92	13.54	1116.57	10.21
$p = 96$	1923.85	15.82	1708.53	14.74
<i>Jaguar</i>	Time (msec)	stddev	Time (msec)	stddev
$p = 1728$	633.23	4.46	556.78	4.38
$p = 5832$	831.88	7.86	612.77	6.52
$p = 13824$	1034.24	8.73	728.01	6.45
$p = 27000$	1149.27	7.31	876.16	7.98
$p = 46656$	1382.14	9.98	981.49	10.04

The overhead operation consists of five steps: memory allocation of the necessary arrays, sort the **rank** array, find common divisors between process grid size and node size, re-arrange **rank** array, and finally create and distribute the communicators. The sorting function is of the order $O(p \log p)$ where p is the number of processes. Finding the common divisors is done in $O(\sqrt{k} \cdot \sqrt{p})$, where k is the number of cores on a single node. The re-arranging **rank** array has $O(p)$ time complexity. The sorting step dominates total overhead operation, which means that as long as the time complexity of the application is of an order that is greater than a sorting function, the overhead will decrease with an increase of processes.

Tests shows that the total overhead introduced by our application-level process-to-core re-mapping are about 10% on small grids and less than 1% for the larger grids (see Table 6.2). From the results we can conclude that much of the overhead time is a result of latency time spent in the creation and distribution of the different communicators.

Table 5.2: Two Dimensional Tests on *Alamode* (8 MB Message Size).

Communicator	Time (msec)						
	$p = 16$	stddev	$p = 36$	stddev	$p = 48$	stddev	Θ
2D MPI_Bcast std rank	505.9	6.8	1040	9.7	1052	10.2	
2D MPI_Bcast tiled rank	443.3	5.8	714.4	8.2	853.9	7.9	1.35
2D Pipeline Bcast std rank	634.4	6.8	1371	12.5	1504	18.7	
2D pipeline Bcast tiled rank	513.6	6.2	1103	12.8	1215	10.4	1.24
2D Binom Bcast std rank	621.7	5.4	1048	9.9	1105	11.4	
2D Binom Bcast tiled rank	334.5	5.2	861.4	9.7	933.7	8.7	1.30
2D MPI_Gather std rank	298.0	4.3	409.0	5.2	470.7	5.2	
2D MPI_Gather tiled rank	164.0	4.4	362.7	4.3	421.8	4.3	1.24
2D MPI_Scatter std rank	240.5	4.4	314.2 4.3		324.1	4.4	
2D MPI_Scatter tiled rank	148.0	3.4	221.1	3.8	269.1	3.7	1.37

Table 5.3: Two Dimensional Tests on *RA* (16 MB Message Size).

Communicator	Time (msec)						
	$p = 48$	stddev	$p = 64$	stddev	$p = 94$	stddev	Θ
2D MPI_Bcast std rank	288.5	4.4	279.2	4.8	281.7	6.2	
2D MPI_Bcast tiled rank	250.0	4.5	256.7	4.7	265.2	3.8	1.10
2D Pipeline Bcast std rank	315.4	5.2	324.8	5.1	343.6	6.3	
2D pipeline Bcast tiled rank	238.2	5.3	268.2	4.8	291.3	5.8	1.17
2D Binom Bcast std rank	342.1	6.2	368.4	5.8	392.6	5.7	
2D Binom Bcast tiled rank	297.6	5.6	342.4	5.8	338.7	5.2	1.13
2D MPI_Gather std rank	599.2	7.2	651.5	6.9	667.0	7.2	
2D MPI_Gather tiled rank	353.7	6.1	380.2	6.2	396.1	7.1	1.69
2D MPI_Scatter std rank	856.9	10.2	879.5	9.6	909.7	10.3	
2D MPI_Scatter tiled rank	564.2	6.2	582.9	5.8	590.3	6.1	1.53

Table 5.4: Two Dimensional Tests on *Jaguar* (64 MB Message Size).

Communicator	Time (msec)						
	$p = 20000$	stddev	$p = 30000$	stddev	$p = 40000$	stddev	Θ
2D MPI_Bcast std rank	512.5	6.2	551.5	6.8	598.5	6.8	
2D MPI_Bcast tiled rank	332.8	5.1	335.6	5.4	344.8	6.2	1.64
2D Pipeline Bcast std rank	535.3	6.7	586.4	7.3	718.9	10.5	
2D pipeline Bcast tiled rank	385.1	5.7	471.6	7.1	458.9	6.9	1.40
2D Binom Bcast std rank	703.9	8.3	759.4	8.4	979.8	10.5	
2D Binom Bcast tiled rank	408.7	5.4	495.8	6.8	548.6	8.9	1.68
2D MPI_Gather std rank	1808	10.5	1989	15.8	2375	17.8	
2D MPI_Gather tiled rank	960.5	10.6	1260	10.8	1325	18.7	1.88
2D MPI_Scatter std rank	1703	10.4	1862	12.3	2093	18.4	
2D MPI_Scatter tiled rank	1028	10.4	1032	11.6	1161	12.6	1.76

CHAPTER 6

APPLICATION IMPLEMENTATION

In this chapter, we present the implementation of the algorithms to improve the multi-dimensional communication. We work with two common applications; the Matrix-Matrix multiplication and the N -body application.

6.1 Matrix-Matrix Multiplication

Matrix-Matrix multiplication has application within several areas such as game theory, text and data mining, encryption, graphics, graph and quantum theory [71–76]. The Basic Linear Algebra Subprograms (*BLAS*) define a set of fundamental operations on vectors and matrices which can be used to create optimized higher-level linear algebra functionality [77–79]. There are three levels of BLAS operations:

- **Level 1:** Vector operations such as $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, this includes vector norm functions and scalar dot products for example.
- **Level 2:** Matrix-Vector operations such as $\mathbf{y} = \alpha\mathbf{Ax} + \mathbf{y}$,
- **Level 3:** Matrix-Matrix operations such as $\mathbf{C} = \alpha\mathbf{AB} + \mathbf{C}$

BLAS is frequently used in HPC (High Performance Computing), highly optimized implementations of the BLAS interface have been developed both Intel (the Intel Math Kernel Library, Intel MKL) and AMD (the AMD Core Math Library, ACML), as well other authors; Eigen BLAS, Goto BLAS and OpenBLAS to mention some.

The General Matrix Multiply (GEMM) is a BLAS subroutine which performs Matrix-Matrix multiplication. This includes the `dgemm()` which is the function for double precision. The GEMM calculates the new value of matrix \mathbf{C} based on the matrix-product of matrices \mathbf{A} and \mathbf{B} , and the old value of matrix \mathbf{C}

$$\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$$

here α and β are scalar coefficients and not *Latency* and *bandwidth*. GEMM is an important building block of other numeric software and it is also an important building block for calls to GEMM for larger matrices. By decomposing one or both of the input matrices into block matrices, GEMM can be used repeatedly on the smaller blocks to build up a result for the full matrix. This is the application we will implement [71, 71, 72, 77].

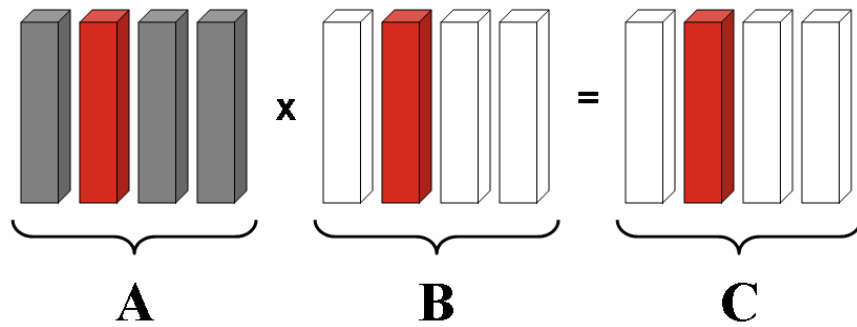


Figure 6.1: Matrix-Matrix Multiplication with a one dimension decomposition of matrices \mathbf{A} , \mathbf{B} and \mathbf{C}

Assume that \mathbf{A} , \mathbf{B} and \mathbf{C} are dense matrices of size $\sqrt{m \cdot p}$, so that each process p holds a block of size m from each matrix. The first step is to decompose the matrices (\mathbf{A} , \mathbf{B} and \mathbf{C}). Let us first consider a one dimensional (columnwise) decomposition. Figure 6.1 shows a one dimension decomposition of the Matrix-Matrix multiplication. The blocks of matrices \mathbf{A} , \mathbf{B} and \mathbf{C} associated with a single process are marked in red; during the computation this process requires not only access to the red blocks but also the complete matrix \mathbf{A} , here shaded gray. As seen each process holds the corresponding columns from \mathbf{A} , \mathbf{B} and \mathbf{C} . The parallel algorithm assigns each process the task of all computations associated with its $\mathbf{C}_{i,j}$, as each process needs all of \mathbf{A} to compute its $\mathbf{C}_{i,j}$, m amount of data is required from each of the $p - 1$ other process in each step, giving us a communication cost of:

$$\begin{aligned}
T_{mm-mult1d} &= (p-1)(\alpha + \beta \cdot m) \\
&\approx p \cdot (\alpha + \beta m)
\end{aligned}$$

where α is system latency and $1/\beta$ is the bandwidth. Each process makes $O(m^{3/2} \cdot \sqrt{p})$ computations, if $m \approx p$ (meaning each process holds a single element) the algorithm will have to transfer one word of data for computational step, which means we cannot expect the one dimension decomposition to be efficient until each process holds a significant size of elements and computation is much larger than the time to transfer m .

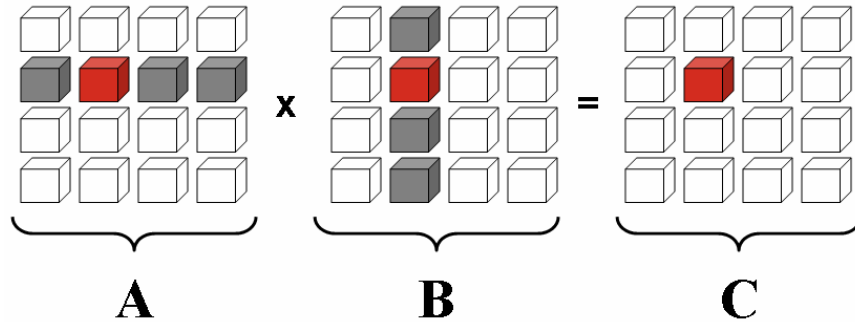


Figure 6.2: Matrix-Matrix Multiplication with a two dimension decomposition of matrices **A**, **B** and **C** (Fox's Algorithm).

Let us now consider a two dimensional decomposition [14, 80, 81]. Figure 6.2 shows an example of the Matrix-Matrix multiplication decomposed in two dimensions, a parallel implementation of *Fox's Algorithm*. The blocks of matrices **A**, **B** and **C** associated with a single process are marked in red, during the computation this process requires the corresponding rows and columns of matrix **A** and **B**, marked in gray. The parallel algorithm assigns each process the task of all computations associated with its $C_{i,j}$, as each process needs the entire row A_i and column B_j to compute its $C_{i,j}$. The amount of data is significantly less than in the one-dimensional decomposition $O(m/\sqrt{p})$, and the cost for communications becomes:

$$\begin{aligned}
T_{mm-mult2d} &= (\sqrt{p} - 1) \left(\frac{\log_2 p}{2} + 1 \right) (\alpha + \beta m) \\
&\approx \frac{\sqrt{p} \log_2 p}{2} (\alpha + \beta \cdot m)
\end{aligned}$$

Algorithm 6.13: Parallel Matrix-Matrix Multiplication based on two-dimensional decomposition of a square Matrix

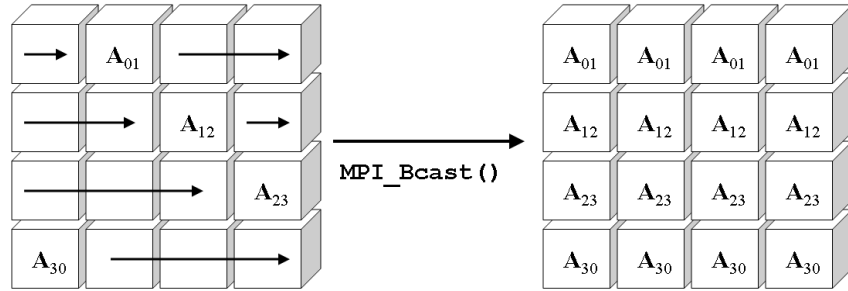
Input: Local sub-matrix **A** and **B**
Output: Local sub-matrix **C**

```

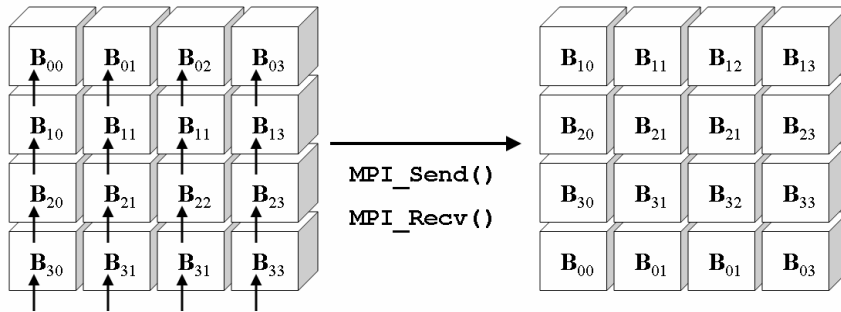
1 B' ← B
2 row ← number of rows in process grid
3 column ← number of columns in process grid
4 my_rank ← local position in the row-communicator
5 for i ← 1 to column do
6   root ← (i + j) mod column
   /* If the process is the current root, broadcast local sub-matrix A
   to rest of row */
7   if my_rank is equal to root then
8     | MPI_Bcast(A, ..., root, row)
9     | A' ← A
10  end
   /* If not receive, temp sub-matrix A from root */
11  else
12    | MPI_Bcast(A', ..., root, row)
13  end
   /* Rotate the sub-matrix B' */
14  MPI_Send(B', ..., neighbor above)
15  MPI_Recv(B', ..., neighbor below)
   /* Multiply the sub-matrices and accumulate in C */
16  C ← C + dgemm(A', B')
17 end
18 MPI_Barrier(MPI_COMMUNICATOR)
19 Collect C in root

```

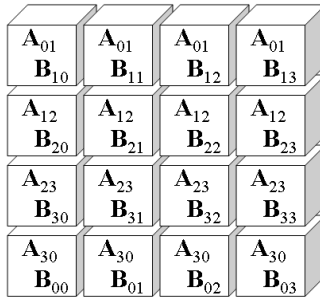
Our implementation uses a standard algorithm with `MPI_Bcast()` and `dgemm()` to multiply two large matrices (see Figure 6.3). It is easy to evaluate this implementation because each step in the multiplication has two distinctive sub-steps; a communication-step (Figure 6.3(a) and Figure 6.3(b)) followed by a calculation-step (Figure 6.3(c)). We first distribute the two matrices, of type `double`, as sub-matrices on a process grid. Then perform the multiplication



(a) Step 1: Broadcast the sub-matrix of \mathbf{A} to processes in the same row



(b) Step 2: Send the sub-matrix \mathbf{B}' to neighbor above



(c) Step 3: Accumulate $\mathbf{A}' \cdot \mathbf{B}'$ into \mathbf{C}

Figure 6.3: Example of one step in of the Parallel Matrix-Matrix multiplication Algorithm.

by broadcasting the sub-matrices and then uses `dgemm()`. By fixing the sub-matrix dimension for all tests to a large size $A \times A$, setting the relationship between the global matrix and processors to $p \cdot A^2$ each `dgemm()` will have the same size as only the number of calls will increase as the size of the global matrix increases. Furthermore, the size of the broadcast will stay constant while the number of calls and the number of members in the communicator will change as the number of processes increase or decrease.

6.1.1 Results

The tests were conducted on the *Alamode*, *RA* and *Jaguar* systems. We fixed the sub-matrix dimension to 2048×2048 , setting the relation between the global matrix and processes to $p \cdot 2048^2$.

Table 6.1: Matrix-Matrix Multiplication, 2048×2048 sub-matrix size

	Communicator			
	Standard		Tiled	
Grid Size	Comm.	Calc.	Comm	Calc.
Alamode				
$p = 16$	5.807 s	16.77 s	4.911 s	16.86 s
$p = 36$	10.20 s	23.66 s	8.736 s	23.72 s
$p = 64$	14.51 s	33.72 s	13.11 s	33.56 s
RA				
$p = 16$	1.559 s	7.634 s	1.073 s	7.666 s
$p = 64$	3.797 s	15.15 s	2.314 s	15.14 s
$p = 144$	6.008 s	22.63 s	3.588 s	22.67 s
Jaguar				
$p = 5184$	117.8 s	134.9 s	54.83 s	135.2 s
$p = 9216$	135.8 s	179.9 s	74.75 s	180.3 s
$p = 20736$	271.7 s	269.7 s	122.5 s	270.3 s

The results confirm that our implementation does not have any impact on the time taken by the calculation step (Table 6.1). They also confirm that there is indeed an improvement in the communication step.

The tests on *Alamode* showed that calculation-communication ratio was about 7:3. The overall improvement was around 5%, with an improvement in communication of 15% (Ta-

ble 6.1).

The ratio between calculation and computation on *RA* was close to $4:1$. The overall improvement was around 8–10%, with an improvement in communication of 40% (Table 6.1).

The calculation-communication ratio on *Jaguar* was $1:1$. There was a 25% overall improvement. The improvement of the communication step alone, was 50% (Table 6.1).

The decreased time for communication, overall, had a positive impact on the complete function. The subsequent improvements were within the expected ranges.

6.1.2 Overhead

These experiments also provided us with the opportunity to measure the *Re-mapping* overhead against an algorithm that is performing some extensive calculation between each communication step. As described in 5.4.6, the overhead consists of five steps: memory allocation of the necessary arrays, sort the **rank** array, find common divisors between process grid size and node size, re-arrange **rank** array, and finally create and distribute the communicators. Where the most of the time is latency time spent in the creation and distribution of the different communicators.

Table 6.2: Tiling function overhead with Matrix-Matrix Multiplication

Grid Size	Overhead	Total time
Alamode		
$p = 16$	2.02 s	23.79 s
$p = 36$	2.12 s	34.58 s
$p = 64$	2.86 s	49.53 s
RA		
$p = 16$	2.02 s	10.75 s
$p = 64$	2.07 s	19.52 s
$p = 144$	2.32 s	288.58 s
Jaguar		
$p = 5184$	2.06 s	193.0 s
$p = 9216$	2.16 s	258.2 s
$p = 20736$	2.51 s	397.4 s

There is a less than 10% overhead for small grids, which is a significant amount, but for larger grids $p > 4096$ the overhead is just 1% which at times is less than the standard deviation for the total run time.

6.2 The N -body Problem

A large number of physical systems can be studied by simulating the interactions between the particles constituting the system. In a typical system each particle influences every other particle, often based on an inverse square law such as Newton's law of universal gravitation or Coulomb's law of electrostatic interaction. Examples of physical systems and simulation that uses the N -body Problem can be found in astrophysics, molecular dynamics and fluid dynamics; the interaction even has applications with robotics and swarm control [14, 19, 82]. Since the simulation involves following the trajectories of motion of a collection of N particles, the problem is termed the N -body problem. Apart from traditional applications in the study of physical systems, some problems in numerical complex analysis and elliptic partial differential equations can also be solved using this approach. Applications of the problem are also found in computer graphics (radiosity methods), where the attempts are to create images by computing the equilibrium distribution of light for complex scene geometries.

It is not possible to solve the equations of motion for a collection of four or more particles in closed form, so iterative methods have been developed to solve (simulate) the N -body problem. These methods compute the force on each particle (body) at each discrete time interval. This information is then used to update the position and velocity of each particle. A straightforward computation of the forces requires $O(N^2)$ work per iteration. The rapid growth with N effectively limits the number of particles that can be simulated by this method.

Several approaches have been used to reduce the complexity per iteration. Some of the techniques include transforming the problem to a position-velocity phase space, imposing a grid on the system of particles and computing cell-cell interactions. Another recent approach is a new class of particle simulation methods that have emerged to solve the N -body

problem. These methods are characterized by an organization of the particles into a hierarchy of clusters, starting from a cluster containing all the particles to clusters containing the individual particles. These methods are usually referred to as hierarchical methods or tree methods. Even though these new methods scale well and some are solid, our focus is on parallel algorithms and channel communication. For this reason will our implementation be a simple straightforward parallelization of the sequential algorithm where computation is performed on every pair of objects, with an $O(N^2)$ time-complexity.

Suppose we are trying to simulate the motion of N -bodies of varying mass, location and velocities. During each iteration of our algorithm we will need to compute the new position and velocity vector of each particle (body), given the positions and masses of all other particles (see Figure 6.4).

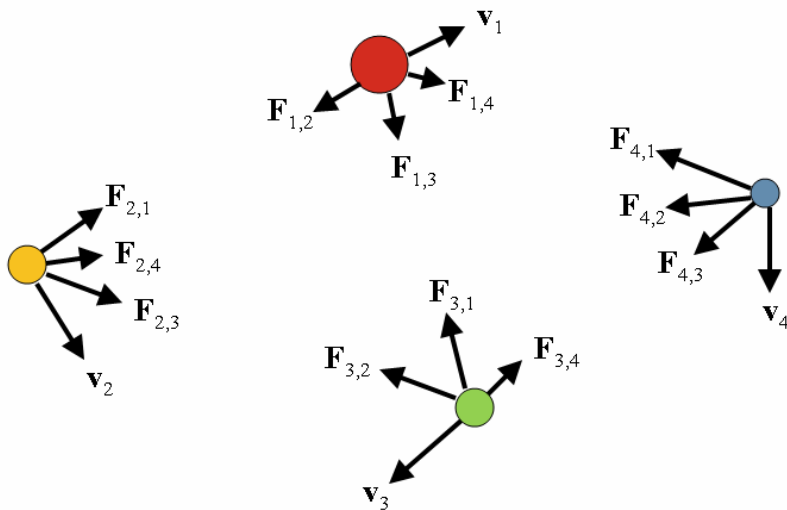


Figure 6.4: N -body problem with four particles, each particle exerting a gravitational on all other particles. The future position of each particle will be determined by the sum of all forces acting on it.

Applying Newton's law of universal gravitation we can determine the gravitational pull between each particle pair:

$$\mathbf{F}_{i,j} = G \frac{m_i \cdot m_j}{|\mathbf{r}_{i,j}|^2} \frac{\mathbf{r}_{i,j}}{|\mathbf{r}_{i,j}|}$$

In Newton's law of universal gravitation $\mathbf{F}_{i,j}$ is the force vector between the two masses (m_i, m_j) , $\mathbf{r}_{i,j}$ is the displacement vector between the center of the two masses and $|\mathbf{r}_{i,j}|$ the magnitude (distance) between them. When considering the whole system of N -particles interacting and applying the second law of motion provides us with the following system of equations:

$$\begin{aligned}
\mathbf{F}_1 &= \mathbf{F}_{1,2} + \mathbf{F}_{1,3} + \cdots + \mathbf{F}_{1,N} = m_1 \cdot \mathbf{a}_1 \\
\mathbf{F}_2 &= \mathbf{F}_{2,1} + \mathbf{F}_{2,3} + \cdots + \mathbf{F}_{2,N} = m_2 \cdot \mathbf{a}_2 \\
&\vdots = \vdots \\
\mathbf{F}_i &= \mathbf{F}_{i,1} + \mathbf{F}_{i,2} + \cdots + \mathbf{F}_{i,N} = m_i \cdot \mathbf{a}_i \\
&\vdots = \vdots \\
\mathbf{F}_N &= \mathbf{F}_{N,1} + \mathbf{F}_{N,2} + \cdots + \mathbf{F}_{N,N-1} = m_N \cdot \mathbf{a}_N
\end{aligned}$$

Focusing on the i^{th} equation using both the definition of acceleration and the equation for Newton's law of universal gravitation we arrive at the following equation for the forces under mutual gravitation:

$$m_i \cdot \mathbf{a}_i = G \sum_{\substack{j=1 \\ j \neq i}}^N \frac{m_i \cdot m_j}{|\mathbf{r}_{i,j}|^2} \frac{\mathbf{r}_{i,j}}{|\mathbf{r}_{i,j}|}$$

Finally for $1 \leq i \leq N$, divide sides by m_i (assuming mass for each particle is constant during the simulation) we get the following system of N second order autonomous vector differential equations:

$$\mathbf{a}_i = G \sum_{\substack{j=1 \\ j \neq i}}^N \frac{m_j}{|\mathbf{r}_{i,j}|^2} \frac{\mathbf{r}_{i,j}}{|\mathbf{r}_{i,j}|}$$

We are not interested in solving the system, but to implement it to recalculate the position and velocity for each particle in each discrete time-step. We have one task per particle, and in order for this task to compute the new location and velocity for the particle, it must know the mass and location of all other particles. Let N be the number of particles, and p the number of processes. It does not matter which particle is assigned to which process (as all

processes need to know the location of each particle in order to finish their computations), so we can simply divide the tasks (particles) evenly on all ps , with each process having N/p particles.

6.2.1 Communication

The `MPI_Gather()` function collects all datasets distributed among all members in the group (communicator) into a single process (see Figure 6.5). This solves only half our problem, as we want all the process to have all the data. We could run gather and let all processes be *root*. We could also gather all the data into a single process and then broadcast the concatenated vector to all process. This is close to what an `MPI_Allgather()`, a function where every process in the group at the end of communication step has a copy of the entire data set (see Figure 6.6).

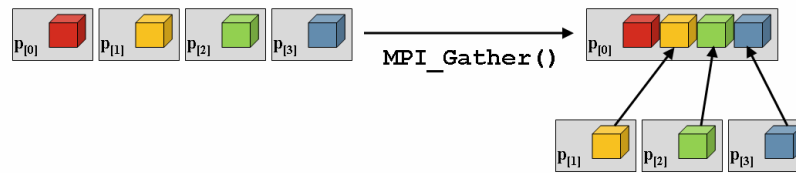


Figure 6.5: The `MPI_Gather()` communication concatenates all the data into a single process.

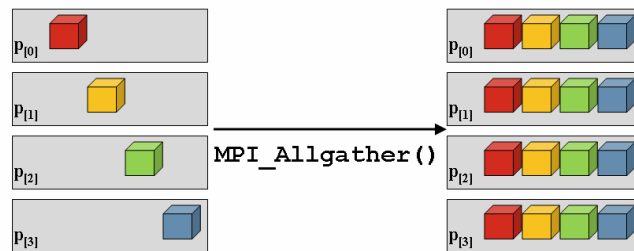


Figure 6.6: The `MPI_Allgather()` communication concatenates all the data on all processes.

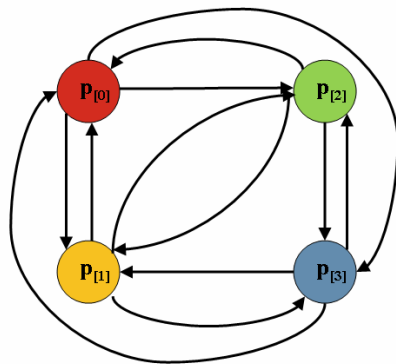
In our case we like to update the location (as we assume mass of each particle is constant during the whole simulation it can be distributed at the beginning of the simulation) of every particle in every step (see Algorithm 6.14). There are several ways we could do this;

Algorithm 6.14: Parallel N -body Algorithm

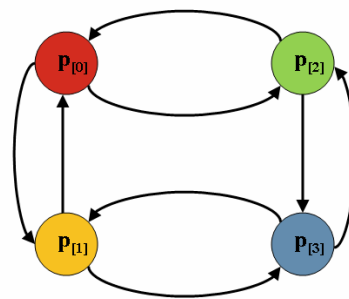
Input: A vector of the current state of all local particles
Output: A vector of the final state of all particles

```
1  $time \leftarrow$  amount of discrete time-steps
2  $p \leftarrow$  number of processes
3  $N \leftarrow$  number of particles
4  $local\_n \leftarrow N/p$ 
   /* Create a vector with the local state (Position, Velocity and Force)
   of all local particles. */
5 local_state  $\leftarrow$  the state of all local particles
   /* Create a vector with the masses of all particles. */
6 mass  $\leftarrow$  the mass of all particles
7 for  $step \leftarrow to\ time$  do
8    $t \leftarrow step \cdot \Delta t$ 
9   MPI_Allgather( $position, local\_position, \dots, communicator$ ) /* Create a vector
   with the masses of all particles. */
10  for  $local\_particle \leftarrow 0\ to\ local\_n$  do
   /* Update the state (Force, Position and Velocity) of each local
   particle. */
11  Update_Local_Particle(local_state, mass, position)
12  end
13 end
14 MPI_Allgather( $state, local\_state, \dots, communicator$ )
15 Display the final state for all particles
```

two were mentioned above. Another approach is to create a channel between every pair of processes (see Figure 6.7(a)). At each step of the communication each process sends its vector to another task. After $p - 1$ communication steps will each process have the positions of all the other particles, and is ready to begin the calculation. A faster way to accomplish communication would be to allow each process to receive on one channel and send on another. Assume we have two processes; in step one they exchange the information and we are done. What if there are more processes? In step one let all even process exchange their data with the next higher rank. Process $p_{[0]}$ and $p_{[1]}$ will both have all the data for both their own particles and their neighbors; $p_{[2]}$ and $p_{[3]}$ will have their own and their neighbors data. If process $p_{[0]}$ exchanges the accumulated data with $p_{[2]}$ and $p_{[1]}$ exchanges its accumulated data with $p_{[3]}$, all processes will have all the data. The process communication graph in Figure 6.7(b) shows an example of a hypercube network, a common implementation of *All-to-All* data exchange. Each communication operation requires $\log_2 p$ communication steps. In the first the message will have the size of $m = N/p$, in the second $2m = 2N/p$ etc.



(a) `MPI_Allgather()` through all pair exchange.



(b) `MPI_Allgather()` through hypercube exchange.

Figure 6.7: Two different implementations of `MPI_Allgather()` communication.

Using the Hockney Model (see Equation 3.1) we are able to estimate the communication time for each iteration to:

$$T(m) = \sum_{i=1}^{\log_2 p} (\alpha + 2^{i-1} \cdot m \cdot \beta) = \alpha \log_2 p + (p - 1) \cdot m \cdot \beta$$

Each process is responsible for performing the update of the *state* (Position, velocity and force) on $m = N/p$ particles, assuming the time necessary to perform these calculations is γ . The total execution time per iteration then becomes:

$$T(m) = \alpha \log_2 p + m \cdot ((p - 1) \cdot \beta + \gamma)$$

6.2.2 Results

Even if the data is multi-dimensional (the state of a particle is described by its three dimensional location, and its velocity vector), the communication is single dimensional. We will therefore implement the *Channel Aware* communication and compare the results against the results from the standard *MPI* implementation.

The tests were conducted on the new *Alamode*-lab, and the *RA* systems. We fixed the number of particles per core to 250, setting the relation between the global data and processors to $N = p \cdot 250$. Further, we fixed the number of discrete *time steps* to 1000 and the Δt to 0.01 *s*.

We had to use the new *Alamode*-lab as the machines from the old lab had been disbursed to users all around the campus. We tried to locate 16 of the old machines and get permission to use them (to have 64 cores), but we noticed during the initial testing that the external bandwidth was erratic, due to the fact it is shared network. We had to abandon that idea and to use the machines in the new lab. The new system created some new problems. The machines are *Intel i7* processors, each has four physical cores, but they also implements *Intel's Hyper-Threading Technology* with two threads per core. Even though *Hyper-Threading* increases the performance (with as much as 15-30% compared to non-threaded processor by *Intel's* claim [83]), it does increase cache thrashing. The *Hyper-Threading* can be disabled in the BIOS, but this is not a possible solution when running tests on the machines in the lab. We only compare results against each other within the system, but we still like to minimize

the impact from threading. We therefore decided to run the tests during hours when the resources in the lab have a low utilization. Further, we requested all the resources on a machine, but only utilized 4 cores for the tests. Even though this is no guarantee for not allocating *Hyper-Threaded* cores instead of physical cores, initial testing showed benchmark peak performance with this set-up.

The network characteristics on the new machines were similar to the characteristics on the old machines ($\beta_E \approx 120$ MB/s and $\beta_I \approx 1100$ MB/s).

Table 6.3: Results from the N -body tests, $N/p = 250$ time steps = 1000 and $\Delta t = 0.01$

	Communicator			
	Standard		Channel Aware	
Grid Size	Comm.	Calc.	Comm	Calc.
<i>Alamode</i>				
$p = 16$	0.883 s	15.08 s	0.864 s	15.02 s
$p = 36$	1.926 s	23.18 s	1.932 s	23.54 s
$p = 64$	3.384 s	45.35 s	3.346 s	43.57 s
<i>RA</i>				
$p = 16$	0.264 s	16.91 s	0.271 s	16.73 s
$p = 64$	0.953 s	21.54 s	0.946 s	22.18 s
$p = 144$	3.101 s	312.4 s	3.062 s	314.6 s

Once again our implementation has no impact on the time taken by the calculation step (Table 6.3). They also show that there is no improvement in the communication step.

The tests on *Alamode* showed that calculation-communication ratio was about $12:1$. Any change in performance was within the range of a statistical error $\sigma \approx 1.00$ s (Table 6.3).

The ratio between calculation and computation on *RA* was close to $20:1$. The overall improvement had no statistical significance $\sigma \approx 0.80$ s (Table 6.3).

The results and the experiments prove that there are both communication patterns and agglomeration patterns that our algorithm is unable to improve, even though we take the hardware topology into account.

CHAPTER 7

CONCLUSION

This chapter summarizes the dissertation and describes future research directions and opportunities.

7.1 Summary

This dissertation made four contributions which include:

1. The design and implementation of an Extended Ping-Pong Test. This is a benchmark that is more accurate than the current one and provides information about the network characteristics both internally between pairs of cores on the same node, and externally between pairs of cores on different nodes; which to our knowledge has not been performed by any current benchmark.
2. The development of a re-mapping algorithm for single node. We extended the work done on non-homogeneous clusters and implemented it into working algorithms for a single node using unmodified standard MPI communicators. When we apply our application-level multicore-aware process-to-core re-mapping scheme at runtime, we are able to show better performance than the current algorithms for communication on binomial trees.
3. The development of an algorithm to improve the multidimensional communication time by as much as $\Theta(\frac{1}{d}ck^{\frac{d-1}{d}})$ over the default mapping. Our tiling algorithm automatically creates and re-maps the communicators to better arrange the cores in the process grid such that the amount of External (*Inter*) communication from each node is minimized.
4. The implementation of the tiling and re-mapping algorithm in two commonly used applications. We showed that our algorithms have the ability to improve the runtime of a parallel implementation of Matrix-Matrix Multiplication. We did this using

our tiling algorithm and the standard functions and libraries in MPI and BLAS. We also implemented our work in an N -body application. Our work showed that there are applications where our algorithms have no impact on the run-times. The N -body application uses *All-to-All* communication but both the tiling and channel aware communication algorithms have little or no impact on that type of communication.

7.2 Future Work

From the results of our research we can expect that as the number of cores continue to increase, the benefits of optimizing these multidimensional protocols through utilization of the binomial tree function and topology awareness will only continue to increase as well. It is necessary that MPI implementations keep up with the advances of the hardware so that the effectiveness of the processors, as a whole, may be retained and continually improved. It is also important to pay attention to the non-standard forms of communication. As the results have thus far proved favorable, we plan on extending these methods to the differences in *External* communication due to system topology, to include testing of other Multi-dimensional MPI communications. Further, we would like to extend this the idea and implementation of the multidimensional collective communication into the area of fault tolerance and diskless-checkpointing. The idea behind these methods of fault tolerance is to keep data on dedicated processes, where the checkpointing is done through reduction function and often over several dimensions [70, 84–89]. It seems our algorithm will have an impact on the performance of the checkpointing algorithms.

We also plan to streamline the tiling capabilities and take advantage of algorithms and methods in other areas. Our goal is the creation of a highly usable application level interface which bridges the gap between standard MPI implementations and every-day application to increase efficiency of HPC systems and take full advantage of the modern hardware advances.

REFERENCES CITED

- [1] TOP500. Top 500 Supercomputer Sites. <http://www.top500.org>, 2012.
- [2] A.S. Bland, R.A. Kendall, D.B. Kothe, J.H. Rogers, and G.M. Shipman. Jaguar: The World's Most Powerful Computer. *Memory (TB)*, **300**(62):362, 2009.
- [3] Yifeng Cui, Reagan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, Amitava Majumdar, and Thomas Jordan. Enabling very-large scale earthquake simulations on parallel machines. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I, May 27-30, Beijing, China*, pages 46–53. Springer-Verlag, 2007.
- [4] Yifeng Cui, Reagan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, and Thomas Jordan. Toward petascale earthquake simulations. *Acta Geotechnica*, **4**(2):79–93, July 2009.
- [5] Christine Task and Arun Chauhan. A model for communication in clusters of multi-core machines. *CoRR*, **abs/0810.2150**, 2008.
- [6] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn. On the efficiency of global combine algorithms for 2-D meshes with wormhole routing. *Journal of Parallel and Distributed Computing*, **24**:191–201, 1995.
- [7] Ching-Tien Ho and S. Lennart Johnsson. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *ICPP*, pages 640–648, 1986.
- [8] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. Bandwidth efficient all-to-all broadcast on switched clusters. *International Journal of Parallel Programming*, **36**(4):426–453, 2008.
- [9] L. Chai, A. Hartono, and D.K. Panda. Designing high performance and scalable mpi intra-node communication support for clusters. In *2006 IEEE International Conference on Cluster Computing*, pages 1–10, 2006.
- [10] A. Mamidala, R. Kumar, D. De, and DK Panda. Mpi collectives on modern multi-core clusters: Performance optimizations and communication characteristics. In *International Symposium on Cluster Computing and the Grid, Lyon, France (May 2008)*. Citeseer, 2008.

- [11] R. Graham and G. Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, 2008.
- [12] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, **35**(3):178–194, 2009.
- [13] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and RC Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance. *Computer Physics Communications*, **97**(1-2):1–15, 1996.
- [14] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1-55860-339-5.
- [15] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing Users’ Group Meeting*, June 1995.
- [16] R. Graham, T. Woodall, and J. Squyres. Open MPI: A flexible high performance MPI. *Parallel Processing and Applied Mathematics*, pages 228–239, 2006.
- [17] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/openMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 February 2009*, pages 427–436. IEEE Computer Society, 2009.
- [18] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann. Elsevier Science & Technology, 2011. ISBN 9780123742605.
- [19] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill, 2003. ISBN 9780072822564.
- [20] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, **19**(13):1749–1783, 2007.
- [21] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walker. MPI: The complete reference, 1996.

- [22] L. Chai, Q. Gao, and D.K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *2007 Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007)*, pages 471–478. IEEE Computer Society, 2007.
- [23] G. Hager, G. Jost, and R. Rabenseifner. Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In *Proceedings of the Cray Users Group Conference*, pages 4–7. Citeseer, 2009.
- [24] M.S. Wu, R.A. Kendall, and K. Wright. Optimizing collective communications on smp clusters. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 399–407. IEEE, 2005.
- [25] Rosa Filgueira, David E. Singh, Juan Carlos Pichel, Florin Isaila, and Jesus Carretero. Data locality aware strategy for two-phase collective I/O. In *High Performance Computing for Computational Science – 8th International Conference (8th VECPAR’08)*, volume **5336** of *Lecture Notes in Computer Science (LNCS)*, pages 137–149, Toulouse, France, 2008. Springer-Verlag (New York).
- [26] J. Zhang, J. Zhai, W. Chen, and W. Zheng. Process mapping for mpi collective communications. *Euro-Par 2009 Parallel Processing*, pages 81–92, 2009.
- [27] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar K. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *IPDPS*, page 84. IEEE Computer Society, 2003.
- [28] T. Ma, G. Bosilca, A. Bouteiller, and J.J. Dongarra. HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *IPDPS*. IEEE Computer Society, 2012.
- [29] T. Ma, G. Bosilca, A. Bouteiller, and J.J. Dongarra. Locality and Topology aware Intra-node Communication Among Multicore CPUs. In *EuroMPI 2010: Proceedings of the 17th EuroMPI conference, September 12–15, Stuttgart, Germany*. LNCS, 2010.
- [30] Abhinav Bhatele. Application specific topology aware mapping and load balancing for three dimensional torus topologies. Master’s thesis, Dept. of Computer Science, University of Illinois, December 2007.
- [31] Abhinav Bhatele. *Automating Topology Aware Mapping for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois, August 2010.
- [32] Krishna Chaitanya Kandalla, Hari Subramoni, Gopalakrishnan Santhanaraman, Matthew J. Koop, and Dhabaleswar K. Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *IPDPS*, pages 1–8. IEEE, 2009.

- [33] Krishna Kandalla, Hari Subramoni, Abhinav Vishnu, and Dhabaleswar K. Panda. Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.
- [34] J.L. Traff. Implementing the mpi process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, page 28, nov. 2002. doi: 10.1109/SC.2002.10045.
- [35] Traff and Ripke. Optimal broadcast for fully connected networks. In *International Conference on High Performance Computing and Communications (HPCC), LNCS*, volume 1, 2005.
- [36] Jesper Larsson Traff and Andreas Ripke. An optimal broadcast algorithm adapted to SMP clusters. In Beniamino Di Martino, Dieter Kranzlmuller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface (12th PVM/MPI'05)*, volume **3666** of *Lecture Notes in Computer Science (LNCS)*, pages 48–56. Springer-Verlag (New York), Sorrento, Italy, September 2005.
- [37] Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected processor-node networks. *J. Parallel Distrib. Comput*, **68**(7):887–901, 2008.
- [38] Yuri Breitbart, Minos N. Garofalakis, Ben Jai, Cliff Martin, Rajeev Rastogi, and Avi Silberschatz. Topology discovery in heterogeneous ip networks: the netinventory system. *IEEE/ACM Trans. Netw.*, **12**(3):401–414, 2004.
- [39] Joshua Lawrence and Xin Yuan. An mpi tool for automatically discovering the switch level topologies of ethernet clusters. In *22nd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS'08), Apr. 17–18, Miami, Florida USA*, pages 1–8. DBLP, 2008.
- [40] S.S. Vadhiyar, G.E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3. IEEE Computer Society, 2000.
- [41] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH.
- [42] C. Task and A. Chauhan. A model for communication in clusters of multi-core machines. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 356–357. IEEE, 2008.

- [43] Bibo Tu, Jianping Fan, Jianfeng Zhan, and Xiaofang Zhao. Performance analysis and optimization of mpi collective operations on multi-core clusters. *The Journal of Supercomputing*, **60**(1):141–162, 2012.
- [44] A. Faraj and X. Yuan. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402. ACM, 2005.
- [45] Amith R. Mamidala, Rahul Kumar, Debraj De, and Dhabaleswar K. Panda. Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *CCGRID*, pages 130–137. IEEE Computer Society, 2008.
- [46] Rajesh Nishtala and Katherine Yelick. Optimizing collective communication on multicores. In *Proc. HotPar '09 (1st USENIX Workshop on Hot Topics in Parallelism), USB Data Stick*, Berkeley, CA, March 2009. Usenix Assoc. UC, Berkeley.
- [47] William Gropp, Ewing L. Lusk, Nathan E. Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, **22**(6):789–828, 1996.
- [48] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 353–377, 2004.
- [49] Ernie Chan, Robert A. van de Geijn, William Gropp, and Rajeev Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (11th PPOPP'2006), ACM SIGPLAN Notices*, pages 2–11, New York, New York, USA, 2006. ACM SIGPLAN 2006.
- [50] Matthias Kühnemann, Thomas Rauber, and Gudula Rünger. Optimizing MPI collective communication by orthogonal structures. *Cluster Computing*, **9**(3):257–279, 2006.
- [51] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *IJHPCA*, **19**(1):49–66, 2005.
- [52] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Full bandwidth broadcast, reduction and scan with only two trees. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, (PVM/MPI'07), Sep. 30 – Oct. 3, Paris, France*, volume **4757**, pages 17–26. LNCS, 2007.

- [53] Rajeev Thakur and William Gropp. Improving the performance of collective operations in MPICH. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *PVM/MPI*, volume **2840** of *Lecture Notes in Computer Science*, pages 257–267. Springer, 2003.
- [54] Daniel G. Chavarría-Miranda, Jarek Nieplocha, and Vinod Tipparaju. Topology-aware tile mapping for clusters of SMPs. In *Conf. Computing Frontiers*, pages 383–392. ACM, 2006.
- [55] Richard Black, Austin Donnelly, and Cédric Fournet. Ethernet topology discovery without network assistance. In *ICNP*, pages 328–339. IEEE Computer Society, 2004.
- [56] Bruce Lowekamp, David R. O’Hallaron, and Thomas R. Gross. Topology discovery for large ethernet networks. *SIGCOMM*, **31**(4):237–248, 2001.
- [57] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *Proceedings of the 22nd annual international conference on Supercomputing, ICS ’08*, pages 94–103, New York, NY, USA, 2008. ACM.
- [58] Ahmad Faraj, Sameer Kumar, Brian Smith, Amith Mamidala, and John Gunnels. Mpi collective communications on the blue gene/p supercomputer: Algorithms and optimizations. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects, HOTI ’09*, pages 63–72, Washington, DC, USA, 2009. IEEE Computer Society.
- [59] G. Almasi, R. Bellofatto, J. Brunheroto, C. Caçcaval, J. Castañós, L. Ceze, P. Crumley, C. Erway, J. Gagliano, D. Lieber, et al. An overview of the blue gene/l system software organization. *Euro-Par 2003 Parallel Processing*, pages 543–555, 2003.
- [60] G. Almási, C. Archer, J. G. Castañós, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, **49** (2/3):393–406, 2005.
- [61] M. Banikazemi, V. Moorthy, and D.K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Parallel Processing, 1998. Proceedings. 1998 International Conference on*, pages 460–467. IEEE, 1998.
- [62] D. Patterson. The trouble with multi-core. *IEEE Spectrum*, **47**(7):28–32, 2010.

- [63] M. Barnett, S. Gupta, D. G. Payne, and L. Shuler. Interprocessor collective communication library (InterCom). In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 357–364, pub-IEEE:adr, 1994. IEEE Computer Society Press.
- [64] P.B. Bhat, CS Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, **63**(3):251–263, 2003.
- [65] J. Lawrence and X. Yuan. An mpi tool for automatically discovering the switch level topologies of ethernet clusters. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.
- [66] Bin Jia. Process cooperation in multiple message broadcast. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User’s Group Meeting, (PVM/MPI’07), Sep. 30 – Oct. 3, Paris, France*, volume **4757**, pages 27–35. LNCS, 2007.
- [67] C. Karlsson, T. Davies, Chong Ding, Hui Liu, and Zizhong Chen. Optimizing Process-to-Core Mappings for Two Dimensional Broadcast/Reduce on Multicore Architectures. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 404–413, sept. 2011. doi: 10.1109/ICPP.2011.26.
- [68] Pitch Patarasuk, Ahmad Faraj, and Xin Yuan. Pipelined broadcast on ethernet switched clusters. In *IPDPS*. IEEE, 2006.
- [69] Pitch Patarasuk, Xin Yuan, and Ahmad Faraj. Techniques for pipelined broadcast on ethernet switched clusters. *J. Parallel Distrib. Comput.*, **68**(6):809–824, 2008.
- [70] Zizhong Chen and Jack Dongarra. Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Transactions on Computers*, **58**:1512–1524, 2009.
- [71] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.*, **39**(5):575–582, September 1995.
- [72] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. *Euro-Par 2011 Parallel Processing*, pages 90–109, 2011.
- [73] J. Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, **12**(3):335–342, 1989.

- [74] James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Softw. Pract. Exper.*, **27**(9):995–1012, 1997.
- [75] Amin Shokrollahi. Ldpc codes: An introduction. In *in Coding, Cryptography, and Combinatorics, Progress in Computer Science and Applied Logic*, volume **23**, pages 85–110. Birkhäuser Verlag, 2004.
- [76] R.A. Brualdi and H.J. Ryser. *Combinatorial matrix theory*. Cambridge Univ Pr, 1991.
- [77] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, **32**:866, 2011.
- [78] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 77. ACM, 2011.
- [79] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. Performance impact of process mapping on small-scale smp clusters - a case study using high performance linpack. In *IPDPS*. IEEE Computer Society, 2002.
- [80] A. Vretblad. *Algebra och kombinatorik*. Liber, 1985.
- [81] D.E. Knuth. *The Art of Computer Programming Volumes 1-3 Boxed Set*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.
- [82] W.M. Spears and D.F. Spears. *Physicomimetics: Physics-Based Swarm Intelligence*. Springer, 2012. ISBN 9783642228032.
- [83] Hong Wang, Perry H. Wang, Ross Dave, Scott M. Ettinger, Hideki Saito, Milind Girkar, Steve Shih wei Liao, and John P. Shen. Speculative precomputation: Exploring the use of multithreading for latency. *Intel Technology Journal*, **06**(01):22–35, 2002.
- [84] J. S. Plank. An overview of checkpointing in uniprocessor and distributed systems focusing on implementation and performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Knoxville, TN 27996 USA, 1997.
- [85] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, **PDS-9**(10):972–986, 1998.

- [86] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Building fault survivable mpi programs with ft-mpi using diskless checkpointing. Technical Report UT-CS-04-540, University of Tennessee Computer Science Department, Knoxville, TN, USA, 2004.
- [87] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '05), June 14 - 17, Chicago, IL, USA*, pages 213–223. ACM, 2005.
- [88] Julien Langou, Zizhong Chen, George Bosilca, and Jack Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM J. Sci. Comput.*, **30**(1): 102–116, 2007.
- [89] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164. ACM, 2008.
- [90] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.
- [91] Richard L. Graham, Galen M. Shipman, Brian Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *CLUSTER*. IEEE, 2006. ISBN 1-4244-0328-6.
- [92] Johnsson and Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEETC: IEEE Transactions on Computers*, **38**, 1989.
- [93] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. *ACM Sigplan Notices*, **34**(8):131–140, 1999.
- [94] Craig A. Lee. Topology-aware communication in wide-area message-passing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (10th PVM/MPI'03)*, volume **2840** of *Lecture Notes in Computer Science (LNCS)*, pages 644–652. Springer-Verlag (Berlin/New York), Venice, Italy, 2003,.
- [95] P. Liu and T.H. Sheng. Broadcast scheduling optimization for heterogeneous cluster systems. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 129–136. ACM, 2000.

- [96] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D.G. Payne, and J. Watts. Inter-processor collective communication library (intercom). In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 357–364. IEEE, 1994.
- [97] M. Barnett and D. Payne. others. optimal broadcasting in mesh-connected architecture. Technical report, Tech. Rep. TR-91-38, CS Dept., The University of Texas at Austin, 1991.

APPENDIX - DESCRIPTION OF MPI-FUNCTIONS

MPI is a portable message-passing system designed to function on a variety of different parallel computers. The strength of MPI lays in its language-independent communications protocol which allows for both point-to-point and collective communication. It provides a simple, easy-to-use interface for the basic user making high-performance message-passing operations available on advanced machines [14, 16, 18–21, 47, 48, 90].

A.1 Point-to-Point Communication

The most basic communication mechanism of MPI is the transmission of data between a pair of processes, one side sending, the other, receiving. MPI provides both blocking and non-blocking send and receive functions [14, 18–21].

A.1.1 Blocking Communication

`MPI_SEND()` makes a standard blocking send. The send buffer is a *count* of consecutive entries of the type indicated by the *datatype* beginning with the entry defined by *buf*. These types are MPI datatypes but have a corresponding type in the host language (Fortran or C). The *count* is allowed to be zero, meaning an empty message. The *comm* argument defines which communicator should be used to send the message. The *dest* specifies which process within the communicator is the intended receiver.

`MPI_SEND(buf, count, datatype, dest, tag, comm)`

IN	buf	address for the first element in send buffer
IN	count	number of elements in the send buffer
IN	datatype	the type of each element (see Table A.1)
IN	dest	the rank of the receiver
IN	tag	a message tag
IN	comm	the communicator

Table A.1: The basic MPI datatypes and the corresponding C types.

MPI datatype	C datatype
<code>MPI_CHAR</code>	signed char
<code>MPI_SHORT</code>	signed short int
<code>MPI_INT</code>	signed int
<code>MPI_LONG</code>	signed long int
<code>MPI_UNSIGNED_CHAR</code>	unsigned char
<code>MPI_UNSIGNED_SHORT</code>	unsigned short int
<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_UNSIGNED_LONG</code>	unsigned long int
<code>MPI_FLOAT</code>	float
<code>MPI_DOUBLE</code>	double
<code>MPI_LONG_DOUBLE</code>	long double
<code>MPI_BYTE</code>	
<code>MPI_PACKED</code>	

Note: the datatypes `MPI_BYTE` and `MPI_PACKED` have no corresponding type in any of the host languages (Fortran or C see Table A.1). A value of type `MPI_BYTE` consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. `MPI_PACKED` is a derived datatype in short it allows the user to “send” a message into a memory buffer and then transmit this buffer to the receiving machines buffer.

The communicator is an object that defines the communication domain. A communication domain allows processes in a group to communicate with each other, or to communicate with processes in another group. Processes in a group are ordered and identified by their rank, grouped together in a **rank** vector. Processes may participate in several communication domains; distinct communication domains can have partially or completely overlapping groups of processes. Each communication domain supports a disjoint stream of communications. Thus, a process may be able to communicate with another process via two distinct communication domains, using two distinct communicators. The same process may be identified by a different rank in the two domains; communications in the two domains do not interfere [14, 18–21].

`MPI_RECV()` makes a standard blocking receive. The receive buffer is a storage of *count* consecutive entries of the type indicated by the *datatype*. A message can be received if its *source*, *tag* and *comm* values matches those of the message. It is possible to accept messages from any sender through the wildcard `MPI_ANY_SOURCE`, a receiver can also accept messages with any *tag*, using the wildcard `MPI_ANY_TAG`.

`MPI_RECV(buf, count, datatype, source, tag, comm, status)`

OUT	buf	address for the first element in receive buffer
IN	count	upper limit of the amount of elements to receive
IN	datatype	the type of each element
IN	source	the rank of the sender
IN	tag	a message tag
IN	comm	the communicator
OUT	status	the return status

It should be noted that there is no wildcard for the communicator (*comm*). It is important to remember that the receive call does not specify the size of the incoming message (only an upper limit) and as the use of wildcard is allowed, there is a need to hold some status values.

These values are returned and stored in the *status* argument, and includes information about the source and tag as well as the count.

The send-receive operation combines, in one call, the blocking sending of one message to a destination and the blocking receiving of another message from a source. The source and destination can but do not need to be the same. This is a useful function when two processes need to exchange data between them, or when a chain of processes need to shift data between them (see Algorithm 6.13).

`MPI_SENDRECV`(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN	sendbuf	address for the first element in send buffer
IN	sendcount	number of elements in the send buffer
IN	sendtype	the type of each send element
IN	dest	the rank of the receiver
IN	sendtag	the send message tag
OUT	recvbuf	address for the first element in receive buffer
IN	recvcount	upper limit of the amount of elements to receive
IN	recvtype	the type of each receive element
IN	source	the rank of the sender
IN	recvtag	a message receive tag
IN	comm	the communicator
OUT	status	the return status

A.2 Non-blocking Communication

The performance of a parallel execution can improve on many systems as they allow overlapping communication and computation. This can be done by threading, allowing one thread to execute while another waits for the communication to complete. Another way to perform this, which often has better performance, is the use of non-blocking communication.

We initiate a send, but we do not wait for the receiver to acknowledge it has received; we instead return later and verify that the data has been copied out of the send buffer. We can do same with receive; we start a receive, but if we don't need the data we continue our execution and we later return and check if the data is stored in the receive buffer. Non-blocking send and receive can be done even if there is a matching receive or not. The blocking and non-blocking functions are compatible with each other, so a non-blocking send can be matched with a blocking receive, etc. The non-blocking communication utilizes *request* objects to identify the communication functions and to link the begin operation with the complete operation [14, 18–21].

`MPI_ISEND(buf, count, datatype, dest, tag, comm, request)`

IN	buf	address for the first element in send buffer
IN	count	number of elements in the send buffer
IN	datatype	the type of each element
IN	dest	the rank of the receiver
IN	tag	a message tag
IN	comm	the communicator
OUT	request	the request handler

It is important that the sender does not access any part of the send buffer after the non-blocking send operation has begun until the request announces a complete send.

Just as we cannot allow the sender to access the send buffer until the request announces that the send is complete, we cannot allow the receiver to access the receive buffer until the request announces the receive is complete.

MPI_IRecv(buf, count, datatype, source, tag, comm, request)

OUT	buf	address for the first element in receive buffer
IN	count	upper limit of the amount of elements to receive
IN	datatype	the type of each element
IN	source	the rank of the sender
IN	tag	a message tag
IN	comm	the communicator
OUT	request	the request handler

So how do we know if the nonblocking operation is complete? We use the MPI_WAIT and MPI_TEST functions. The completion of a send indicates that the sender once again has access to the send buffer, and the completion of a receive indicates that the receive buffer is updated.

MPI_WAIT will return a call when the operation associated with the *request* is completed.

MPI_WAIT(request, status)

INOUT	request	the request handler
OUT	status	the return status

MPI_TEST sets the *flag* to *true* if the operation associated with the *request* is completed. In both the MPI_WAIT and MPI_TEST will *status* hold information regarding the completed operation.

MPI_TEST(request, flag, status)

INOUT	request	the request handler
OUT	flag	non-zero if operation is complete
OUT	status	the return status

A.3 Collective Communications

The MPI collective communications transmit data among all processes in a group specified by the *communicator*. A collective communication is done by having all processes in the group call the communication routine. The transmission is synchronized by the *barrier* without passing data [14, 18, 19, 21]. The following collective communication functions are part of MPI:

- *Barrier* a synchronization across all processes in a group.
- *Global Communication* such as:
 - Broadcast from one to all processes in a group.
 - Gather data from all to a single process in a group.
 - Scatter data from one to all processes in a group.
 - Allgather share all data between all processes in a group
- *Global Reduction* operations such as sum, max, min, etc. This includes functions such as:
 - Reduce from all to a single process in a group.
 - Allreduce from all to one and scattered to all.
 - Scan the data of all processes in a group

A.3.1 All-To-One Communication

`MPI_GATHER` all processes in *comm* sends their send buffer to *root* (note: by definition *root* sends to itself). *Root* collects all the data and store them in *rank* order (see Figure A.1). Note that the receive buffer is ignored by all processes but *root*; this does not necessarily mean that the local system allows that there is no receive buffer allocated for non-root processes.

`MPI_GATHER`(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN sendbuf address for the first element in send buffer

IN	sendcount	number of elements in the send buffer
IN	sendtype	the type of each send element
OUT	recvbuf	address for the first element in receive buffer
IN	recvcount	upper limit of the amount of elements to receive
IN	recvtype	the type of each receive element
IN	root	the rank of the ooot process
IN	comm	the communicator

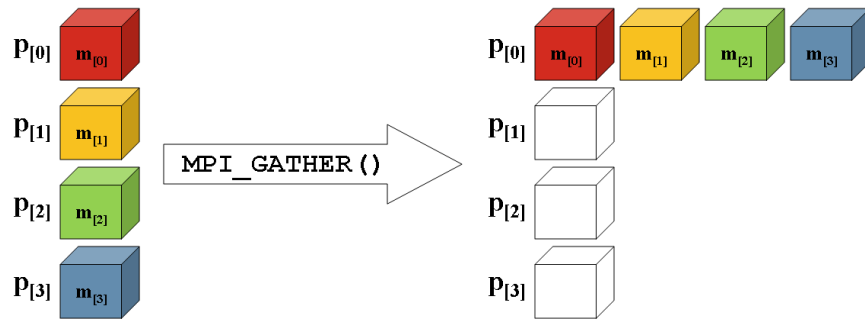


Figure A.1: Example of `MPI_GATHER()` all processes send data to *root*. Each row of boxes represents data locations in one process.

`MPI_REDUCE` combines the operands stored in the memory location referenced by *sendbuf* using the *op*. Note that both the *sendbuf* and the *recvbuf* refer to *count* memory locations of type *datatype* (see Figure A.2). The function has to be called by all processes in the communicator and the arguments have to be the same in all processes.

`MPI_REDUCE`(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer
OUT	recvbuf	address of the receive buffer
IN	count	number of elements in the send buffer
IN	datatype	the type of the send element
IN	op	the reduction operation (see Table A.2)
IN	root	rank of the <i>root</i> process
IN	comm	the communicator

Table A.2: The basic reduction operation that are predefined in MPI.

Name	Meaning
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical AND
<code>MPI_BAND</code>	Bitwise AND
<code>MPI_LOR</code>	Logical OR
<code>MPI_BOR</code>	Bitwise OR
<code>MPI_LXOR</code>	Logical Exclusive OR
<code>MPI_BXOR</code>	Bitwise OR
<code>MPI_MAXLOC</code>	Maximum and the location
<code>MPI_MINLOC</code>	Minimum and the location

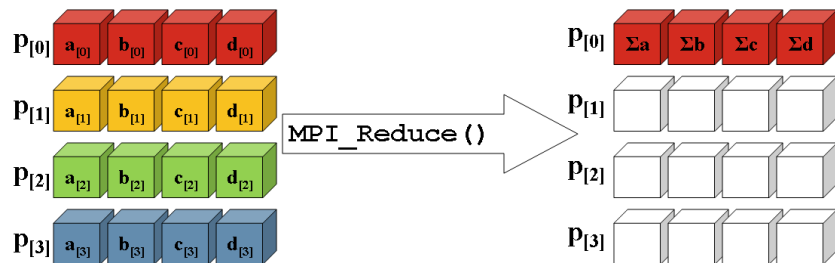


Figure A.2: Example of `MPI_REDUCE()` the combined results are stored in *root*. Each row of boxes represents data locations in one process.

A.3.2 One-To-All Communication

`MPI_SCATTER` the *root* process sends the send buffer to all the processes in *comm* (note: by definition *root* sends to itself). This can be viewed as if the message is split into p equal segments and the i -th segment is sent to the i th process in the communicator (see Figure A.3). Note that the send buffer is ignored by all processes but *root*; this does not mean that the local system allows that there is no send buffer allocated for non-root processes.

`MPI_SCATTER`(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address for the first element in send buffer
IN	sendcount	number of elements in the send buffer
IN	sendtype	the type of each send element
OUT	recvbuf	address for the first element in receive buffer
IN	recvcount	upper limit of the amount of elements to receive
IN	recvtype	the type of each receive element
IN	root	the rank of the oot process
IN	comm	the communicator

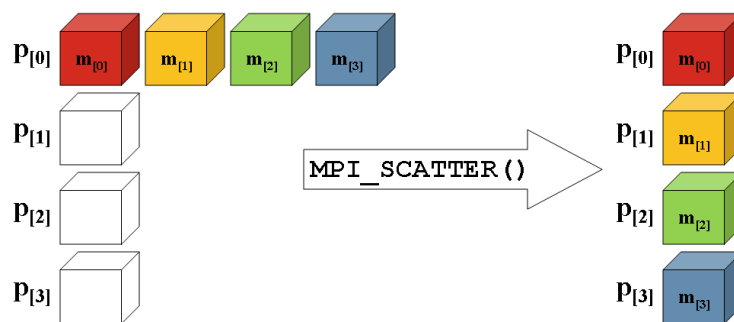


Figure A.3: Example of `MPI_SCATTER()` *root* sends data to all other processes. Each row of boxes represents data locations in one process.

`MPI_BCAST` a message is broadcast from the process with the *root* rank to all processes in the communicator. The argument *root* has to be identical on all processes, and the *comm*

represent a valid communicator. When the function returns the contents of *root*'s buffer is copied to all processes (see Figure A.4)

`MPI_BCAST(buf, count, datatype, root, comm)`

INOUT	<code>buf</code>	address for the first element in the buffer
IN	<code>count</code>	upper limit of the amount of elements to receive
IN	<code>datatype</code>	the type of each element
IN	<code>root</code>	the rank of the root process
IN	<code>comm</code>	the communicator

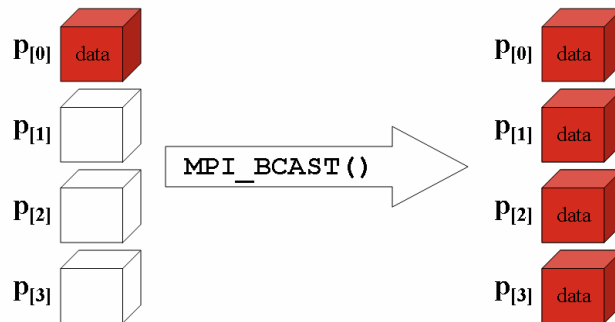


Figure A.4: Example of `MPI_BCAST()` $p_{[0]}$ broadcast a message to all processes in the group. Each row of boxes represents data locations in one process.

A.3.3 All-To-All Communication

`MPI_ALLGATHER` is an `MPI_GATHER` with the difference that all processes, not only *root*, receive the result. The i block of data sent from each process is collected by each process and placed in the i th block of the receive buffer (see Figure A.5). To visualize it assume that each process makes p calls to `MPI_GATHER` with $root = 0, 1, \dots, p - 1$.

`MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`

IN	<code>sendbuf</code>	address for the first element in send buffer
IN	<code>sendcount</code>	number of elements in the send buffer
IN	<code>sendtype</code>	the type of each send element
OUT	<code>recvbuf</code>	address for the first element in receive buffer
IN	<code>recvcount</code>	upper limit of the amount of elements to receive
IN	<code>recvtype</code>	the type of each receive element
IN	<code>comm</code>	the communicator

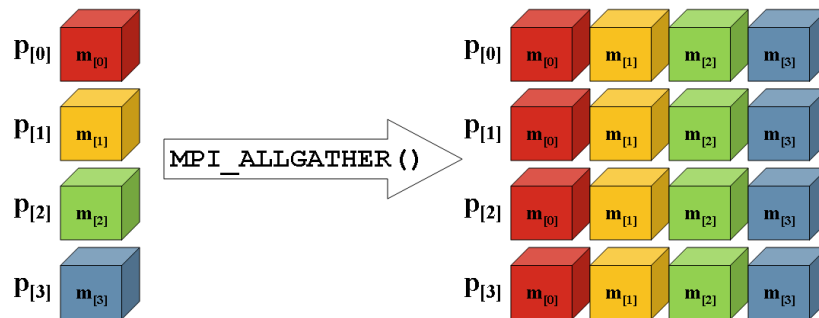


Figure A.5: Example of `MPI_ALLGATHER()` all processes receives the concatenated data. Each row of boxes represents data locations in one process.

`MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)`

IN	<code>sendbuf</code>	address for the first element in send buffer
IN	<code>sendcount</code>	number of elements in the send buffer
IN	<code>sendtype</code>	the type of each send element
OUT	<code>recvbuf</code>	address for the first element in receive buffer
IN	<code>recvcount</code>	upper limit of the amount of elements to receive
IN	<code>recvtype</code>	the type of each receive element
IN	<code>comm</code>	the communicator

`MPI_ALLTOALL` is a collective communication operation in which all processes send a distinct collection of data to every other process. Its effect on process i is to send $sendcount$ elements of $sendtype$ to every process (including self). The first block of $sendcount$ goes to process 0, the second to process 1, etc. Process i will also receive $recvcount$ elements of $recvtype$ from every process (see Figure A.6).

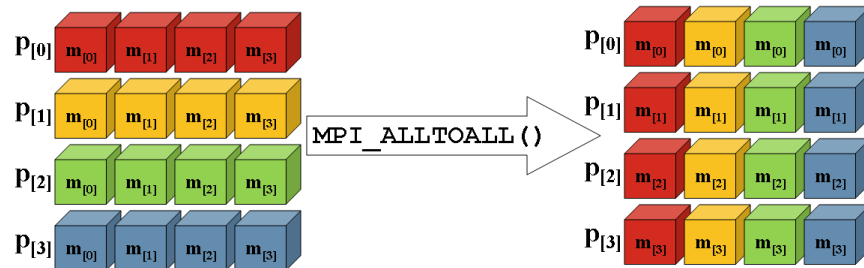


Figure A.6: Example of `MPI_ALLTOALL()` all processes a segment of the data from all other. Each row of boxes represents data locations in one process.

`MPI_ALLREDUCE` is essentially the same as `MPI_REDUCE`, with the difference that the result of the reduction is returned to processes in the communicator and therefore there is no need to define a *root* (see Figure A.7).

`MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)`

IN	sendbuf	address of send buffer
OUT	recvbuf	address of the receive buffer
IN	count	number of elements in the send buffer
IN	datatype	the type of the send element
IN	op	the reduction operation (see Table A.2)
IN	comm	the communicator

`MPI_REDUCE_SCATTER` produces results as if the function first executes an element reduction on the vector $count = \sum_i recvcounts[i]$ in *sendbuf*. After that the resulting vector of results split into p segments where the i th segment holding $recvcounts[i]$ elements is sent and stored in the i th process *recvbuf* (see Figure A.8).

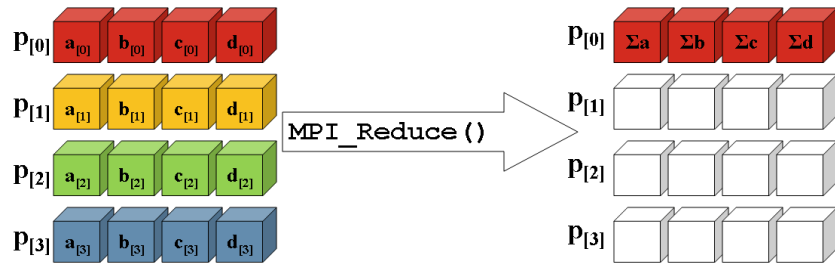


Figure A.7: Example of `MPI_ALLREDUCE()` the combined results are stored in all processes. Each row of boxes represents data locations in one process.

`MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)`

IN	<code>sendbuf</code>	address of send buffer
OUT	<code>recvbuf</code>	address of the receive buffer
IN	<code>recvcounts[]</code>	integer array of number of elements in the receive buffer
IN	<code>datatype</code>	the type of the send element
IN	<code>op</code>	the reduction operation (see Table A.2)
IN	<code>comm</code>	the communicator

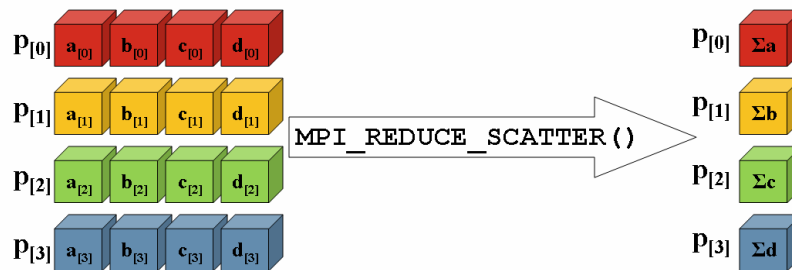


Figure A.8: Example of `MPI_REDUCE_SCATTER()` the combined results of each processor's *sendbuf* using *op*. Scatter the result across all processes in *comm*. Each row of boxes represents data locations in one process.

A.3.4 Other

`MPI_Barrier` is the simplest of all collective communicators; it blocks the caller until all processes in the communicator have called it. This efficiently synchronizes all processes

within a communicator.

`MPI_Barrier(comm)`

IN `comm` the communicator

`MPI_SCAN` conducts a parallel prefix operation; on each process i in `comm`. The result is stored by combining the `sendbuf` on all processes with rank less than i using `op` (see Figure A.9).

`MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)`

IN `sendbuf` address of send buffer

OUT `recvbuf` address of the receive buffer

IN `counts` number of elements in the send buffer

IN `datatype` the type of the send element

IN `op` the reduction operation (see Table A.2)

IN `comm` the communicator

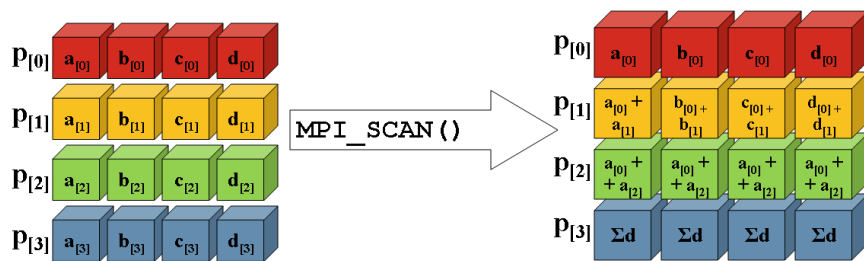


Figure A.9: Example of `MPI_SCAN()` a prefix reduction is performed on all data distributed across the communicator. Each row of boxes represents data locations in one process.

A.4 MPI Groups and Communicators

MPI provides the user with the ability to divide up the processes into different groups. This is a powerful tool; it not only allows the programmer to create subsets of processes that work different tasks within an application. It also enables applications to have collective

communication work on subsets processes. Even if we do not need independent subsets performing different tasks within our application, we might still want the collective operations to work on subsets. It is often desirable to limit the amount of data shared and transferred, as this allows for better scaling [14, 18, 19, 21]. We utilized this opportunity when we created *row* and *column* communicators in the *Matrix-Matrix Multiplication* in 6.1; only those processes in the same *row* shared the information in the `MPI_BCAST()`.

A.4.1 Groups

To enable us to divide the process into different groups, we need some tools. First, we need to define what a group is and how we distinguish it. This is the definition of a group from *MPI: The Complete Reference*:

A group is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer rank. Ranks are contiguous and start from zero. Groups are represented by opaque group objects, and hence cannot be directly transferred from one process to another [21].

Second, we need to be able to manipulate the groups, most important we need the ability to create new groups from existing groups. Before we can create new groups we need a base case as MPI does not provide a mechanism to build a group from scratch, but only from other, previously defined groups. The base group, upon which all other groups are defined, is the group associated with the initial communicator `MPI_COMM_WORLD`. We can from this communicator define our base group with the `MPI_COMM_GROUP` function.

`MPI_COMM_GROUP(comm, group)`

IN	comm	communicator
OUT	group	the group corresponding to the comm

We can now build new groups; these groups can be empty, proper subgroups or the complete original group. This is done by `MPI_GROUP_INCL` or `MPI_GROUP_EXCL`

`MPI_GROUP_INCL(group, n, ranks, newgroup)`

IN	group	the group from which new group is created
IN	n	the size of the <i>rank</i> array
IN	ranks	ranks of the processes that are in the new group
OUT	newgroup	the new group created

`MPI_GROUP_EXCL(group, n, ranks, newgroup)`

IN	group	the group from which new group is created
IN	n	elements in <i>rank</i>
IN	ranks	ranks of the processes NOT in the new group
OUT	newgroup	the new group created

MPI has several other functions defined for group operations, but these three are the most commonly used and they provide the basic functionality to create new groups.

A.4.2 Communicators

The new groups become the basis for the new communicators. The definition of a communicator from *MPI: The Complete Reference* is:

A communicator is an opaque object with a number of attributes, together with simple rules that govern its creation, use and destruction. The communicator specifies a communication domain which can be used for point-to-point communications [21].

There are two important communicator constructors: `MPI_COMM_CREATE` and `MPI_COMM_SPLIT`. The `MPI_COMM_CREATE` creates a new communicator from *group*, no attributes from the original communicator are propagated to the new communicator.

MPI_COMM_CREATE(comm, group, newcomm)

IN	comm	the communicator making the call
IN	group	the group from which to create <i>newcomm</i>
OUT	newcomm	the new communicator

The MPI_COMM_SPLIT partitions the group that is associated with the original communicator into disjoint subgroups. A new communication domain is created for each subgroup and a handle to the representative communicator is returned in *newcomm*.

MPI_COMM_CREATE(comm, group, newcomm)

IN	comm	the communicator making the call
IN	color	the control for subset assignment
IN	key	the rank assignment
OUT	newcomm	the new communicator

An example of MPI_COMM_SPLIT call will be the creation of the *row* communicator. We let p_1 define how many processes are in each row of the process grid, and let p_2 be the number of processes in each column. We then assign each process an integer *column* that defines what column the process belongs to ($p \bmod p_2$). Next we let each process be assigned a *row* that describes what row the process belongs to ($row = p/p_2$). We then call:

```
MPI_COMM_SPLIT( MPI_COMM_WORLD, row, column, rowcomm);
```

This creates p_1 amount of *rowcomm* each with p_2 elements and where the rank in each *rowcomm* is defined by *column*.

Example: assume we have 12 processes, we want to arrange them into four columns $p_2 = 12/4 = 3$ and three rows $p_1 = 12/3 = 4$ (see Table A.3).

If we call MPI_COMM_SPLIT with *row* as the *color* and *column* as *key* the function will create three new communicators: $row_0 = \{a, b, c, d\}$, $row_1 = \{e, f, g, h\}$ and $row_2 = \{i, j, k, m\}$. On the other hand if we call MPI_COMM_SPLIT with *column* as the *color* and *row* as *key*

Table A.3: Example of the arguments for a `MPI_COMM_SPLIT` call on twelve elements.

process	a	b	c	d	e	f	g	h	i	j	k	m
rank	0	1	2	3	4	5	6	7	8	9	10	11
row	0	0	0	0	1	1	1	1	2	2	2	2
column	0	1	2	3	0	1	2	3	0	1	2	3

the function will create four new communicators: $column_0 = \{a, e, i\}$, $column_1 = \{b, f, j\}$, $column_2 = \{c, g, k\}$ and $column_3 = \{d, h, m\}$. As can be seen this is a very powerful tool.