NTNU
Norwegian Univers
Science and Techno

Comparing Different Implementations of MPI on Multi-core Architecture

Andreas Bach

February 1, 2008

Abstract

Multi-core processors are different from the classical supercomputing paradigm of many standalone processors. It has been growing as an industry trend the last few years. More than 20% of the processors of the Top 500 supercomputer list belongs to this family. With this in mind we examine how current implementations of MPI is performing in a multi-core environment. In this project, we test the MPI implementations MPICH, MVAPICH and Open MPI with regards to local resources. The benchmarks range from average times on individual MPI calls, to High-Performance Linpack to benchmark something closer to a "*real-life*" problem. Our experiments indicate that Open MPI performs best results for benchmarks on the bandwidth term, while MVAPICH has the lowest latency on small and medium data sets. MVAPICH also performs best on average for the smallest data sets of the reduce operation in one of our experiments. Which implementation one should choose would depend on application parameters. Note that Open MPI and MVAPICH is respectively up to 45% and 35% faster than MPICH for the 160kilobytes data set.

Preface

This is the report for the work done in TDT4590 Complex Computer Systems, Specialization Project at NTNU by Andreas Bach. The work was started the autumn 2007, assigned by Associate Professor Dr. Anne C. Elster.

I would like to thank Dr. Elster for the support, inspiration and help throughout the project, and especially for enabling resources and flexibility to finish this report.

I would also like to thank UiTø and the staff at the Computer Center for providing free computing hours on Stallo on newly installed hardware.

Trondheim, 1. February 2008

Andreas Bach

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our goal in this project is to compare different implementations of MPI on clustered SMPs. This will be explored with regards to improvements that might exploit technology such as chip multiprocessor(CMP) in a better way than the implementation currently used on our system do.

Using an improved MPI implementation for a given system can make a significant performance impact on a number of applications. Recent studies[7, 8] show that intra-node communication is more common than intuition might suggest. Improving MPI algorithms for multi-core architectures have been shown to improve benchmark execution time by up to 70%. Scalability of this types of improvements to both number of cores and number of total nodes will be explored. Conclusively, there will be some suggestions to what our system will gain the most from.

With the introduction of multi-core processors in supercomputing, a whole new field of optimizations for collective operations is available. While multiprocessor systems has been on the market for many years, sharing of resources closer to the core of the processor is becoming dominant. With this type sharing the possibility to transfer data in a faster manner than the standard MPICH[10] implementation is doing.

As stated by Thakur and Gropp in [32] there are many open issues in MPI, in spite of the second version being over a decade old. The range is broad, and covers everything from basic operations, scalability to collective operations.

1.1 Outline

In section 2 we will explore different types of MPI implementations and their advantages and disadvantages. Section 3 will be devoted to techniques used for discovering the differences. Section 4 contains some usability considerations. The results is presented in section 5, while we will come to some conclusions and recommendations for the system in section 6.

Chapter 2

Background

2.1 Multi-core processors

In the late 1990s, microprocessor performance improved at an overall rate of 50 to 60% per year. A study from 2000, [1] found that this development would not continue by only expanding pipelines, clock rate scaling and capacity scaling. While this conclusion does not take into consideration the possibility of novel and revolutionizing technology, they point to the fact that communication delays on-chip would become significant for global signals. Also, the superscalar paradigm was getting diminishing returns, in particular with regards to that the clock scaling soon would slow down. The projected technology size in this paper was a bit more pessimistic than the actual development has been, expecting the 35nm technology to be available by 2014. Today, Intel plans to reach 16nm technology by 2013, according to [16] with two-year-cycles of reducing technology size[1].

Later, Huh et al.[14] concluded that higher throughput of future processors may be achieved with scaling the number of cores pr chip. Off-chip bandwidth does limit the number of cores, and if the relationship pins pr core continue to decrease, it will severely limit the throughput.

Hennessy and Patterson addresses the trend of moving towards multiprocessing in [12], listing new and reinforcing factors such as a growing interest in servers and server performance, growth in data-intensive applications, that increasing performance on desktops is less important and that replication rather than unique designs provides leverage.

With those predictions of the future of CMP in mind, we look at the history of releases of mainstream multi-core processors.

[1]Measured by the size of a single transistor gate length

2.1.1 Power4

The first processor to go multi-core in the market was IBM's Power4[5]. Two 64-bit PowerPC cores running at 1.3GHz with an unified 8-way set associative L2-cache divided under three separate L2-controllers. Also, an L3 off-chip cache. The Power4-processor was based on a superscalar architecture with speculative out-of-order execution, each core with eight execution units(two integer, two floating point, two load/store, branch unit, and execution unit to perform logical operations on the condition register). The processor was able to issue instructions to each execution unit every cycle, but instruction retirements was limited to five per cycle. Each core had an 64Kbyte L1 instruction cache, and a 32Kbyte L1 data cache with dual ports. Up to eight concurrent data misses and three instruction misses was possible.

2.1.2 Xeon

The Xeon brand from Intel was originally a single-core processor, first released in 1998. In 2005 Intel released their first dual-core server processor. Codenamed Paxville, it had two Xeon cores and a shared L2-cache[18]. The last processor in the Xeon-series to have L3-cache was the 7100 series, codenamed Tulsa[19]. It was also to have on-chip shared L3-cache, and in this way distinguishing it from the Power4. Intel's implementation of SMT(namely HT) enabled the chip to run up to four threads on the two cores. Intel have later released a quad-core processors[17] in the same class, Kentsfield, Clovertown, Tigerton, and latest Penryn. The Kentsfield and Clovertown systems was both a combination of two chips with two cores each, in one packaging(in a 2x2 fashion). This way two cores shared L2-cache, with no L3-cache. All but Penryn is made on a 65nm process, with Penryn at 45nm. The front-side bus(FSB) of the Penryn series is clocked to 1600MHz.

2.1.3 Opteron

The first multi-core processor from AMD was the second generation Opteron, which had two 64-bit Opteron cores, with dedicated L1 and L2-cache. In place of an on-chip L3-cache it had memory-controller. This organization would presumably have less to gain from resource sharing, not sharing any on-chip caches. Any core-to-core communication would have to go via main memory. Each processor chip has its own main memory bank. In multiprocessor settings, adding a CPU increases main memory bandwidth.

2.2 MPI

The Message Passing Interface (MPI)[24] is the *de facto* industry standard for parallel scientific applications running on High Performance Clusters(HPC). Begun in 1992, over 40 organizations participated in discussion and definition of library interface standards for message passing. Version 1.0 of MPI was proposed with the final report of the Message Passing Interface Forum May 5th 1994. Version 1.1 was released June of 1995. Later, some corrections was proposed in MPI 1.2, while completely new functionality was discussed in the definition of MPI 2.0. MPI 2.0 was proposed in 1997 (with dynamic processes, one-sided communication and parallel I/O) and MPI 2.1 under discussion as this report is written.

2.2.1 Collective operations and hardware development

Numerous studies has researched possibilities for exploiting unharvested improvements in both algorithms and hardware. I.E. Multi-threading to exploit thread level parallelism and new all-to-all-algorithms to exploit specific topologies and network layout. Multi-core processors was suggested as early as 2000 by Barroso et al. with their Piranha-processor[21]. The same year, IBM introduced the first multi-core processor in their Power4[5] for the server market. Dual-core processors for desktop environments has been introduced by both Intel and AMD[19, 2]. Later AMD, Intel and IBM has introduced new mainstream multi-core processors in [3], [17] and [15] respectively. There has also been studies as to use the Cell-chip[20] from Sony/IBM for scientific purposes[35], with later elaborate studies to optimize collective operations for heterogeneous multi-core processors[34] by Srinivasan et al.

2.3 MPICH

MPICH[11, 10] is an implementation of MPI developed in collaboration with the MPI standards process to provide the MPI forum[24] feedback on implementation and usability issues. Since MPICH was designed to enable ports to other systems, it is often used as basis for implementations by parallel computer vendors and research groups.

2.4 Improving the Performance of Collective Operations in MPICH

Thakur and Gropp studied[31] improving one specific implementation of MPI, using multiple algorithms depending on message size. Their work was concluded with inclusion in the openly available MPICH implementation version 1.2.6 (current version is 1.2.7). This section will elaborate on some of the modifications they did.

2.4.1 MPI_Allgather

Thaikur and Gropp modified MPI_ALLGATHER from the original ring method to a recursive doubling method(as shown in figure 2.1). In this way they reduced

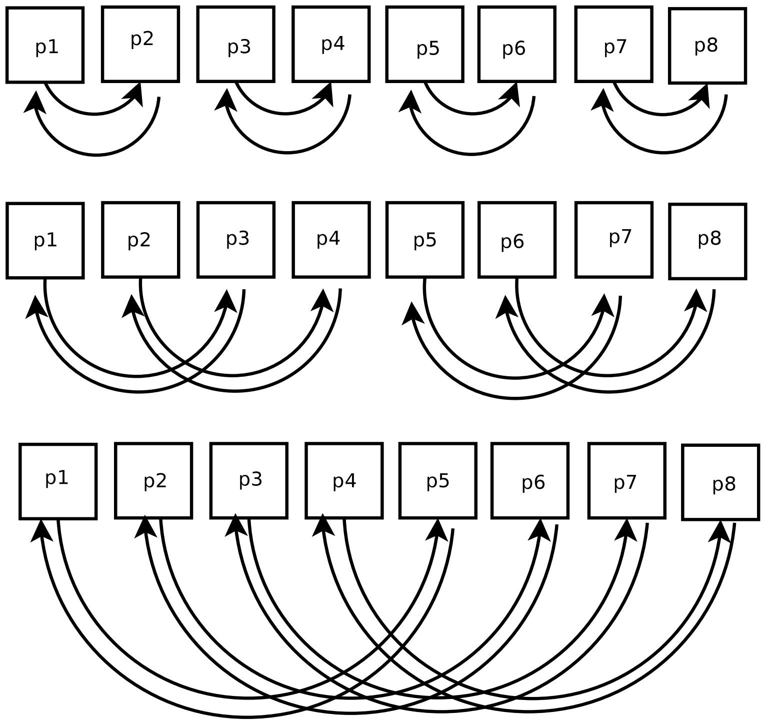

Figure 2.1: Recursive doubling

the time taken from

$$T_{ring} = (p-1)\alpha + \frac{p-1}{p} n\beta$$

where the bandwidth cannot be reduced further, instead reducing latency to lg p

$$T_{rec\_dbl} = \lg p + \frac{p-1}{p} n\beta$$

In experiments they found that recursive doubling was better for short messages, but that the original ring-algorithm was faster for long($\geq 512KB$) messages. The nearest-neighbor communication pattern outperformed recursive doubling which has longer communication paths for this case. Implemented in MPICH is $T_{rec_dbl}$for short- and medium-length ($< 512KB$) and $T_{ring}$for long messages.

2.4.2 MPI_Broadcast

For broadcast the old algorithm in MPICH is the binary tree algorithm(as shown in figure 2.2 except moving from the bottom up). Their improvement for broadcast was implementing an algorithm proposed by Van de Geijn et al. that has a lower bandwidth term[4]. Here one divides and scatter the message among the processes, similar to that of an MPI_Scatter-call. The scattered data is then collected back to all processes similar to an MPI_Allgather. The time taken by the complete broadcast is

$$T_{vandegeijn} = (\lg p + p - 1)\alpha + 2\frac{p-1}{p} n\beta$$

compared to the binary tree algorithm

$$T_{tree} = \lceil \lg p \rceil (\alpha + n\beta)$$

The larger number of processes, the greater expected improvement in performance for the Van de Geijn-algorithm over binary tree. In the new MPICH-implementation, the binary tree is used for short messages($< 12KB$) and the Van de Geijn algorithm for long messages ($\geq 12KB$).

2.4.3 MPI_Reduce

Here, the original algorithm used a binary tree(as shown in figure 2.2) that took

$$T_{tree} = \lceil \lg p \rceil (\alpha + n\beta + n\gamma)$$

which is good for short messages because of the lg p steps. For long messages there exists a better algorithm proposed by Rolf Rabenseifner[28], that reduces the bandwidth term from $n \lg p \beta$ to $2n\beta$ and works in the same manner as Van

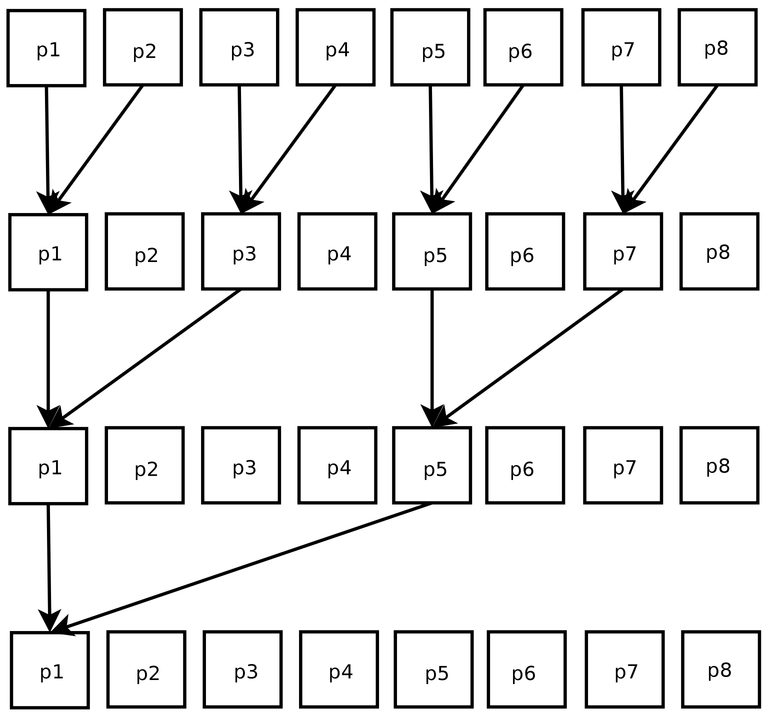

Figure 2.2: Binary tree

de Geijn's algorithm for MPI_Broadcast. The total time for Rabenseifner's algorithm is

$$T_{rabenseifner} = 2\lg p\alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma$$

This algorithm is used for message sizes over 2KB and the binary tree for short messages.

2.5 MVICH

MVICH[26] is a Virtual Interface Architecture(VIA) implementation of MPI, based on MPICH. VIA is an industry standard interface for System Area Networks(that of clusters for example) that provides protected, zero-copy user-space inter-process communication. It is no longer in development, as the funding plan concluded October 2001. Their work was later implemented in MVAPICH by D. K. Panda's group at Ohio State University.

2.6 MVAPICH

MVAPICH[25] is an implementation of MPI based on MPICH and MVICH. It has multiple underlying interfaces, such as OpenFabrics/Gen2 that supports features as SRQ, Shared Memory Collectives, RDMA-based collectives, TCP/IP, multi-rail with advanced scheduling schemes, on-demand connection management and scalable MPD-based startup. It also has support for shared-memory only without using any network, for multi-core servers, desktops, laptops and clusters with serial nodes. New features in the 1.0 beta is OpenFabrics/Gen2-UD that is targeting clusters with multi-thousand cores using InfiniBand.

MVAPICH is especially optimized for intra-node communication by using shared-memory communications, on everything from multi-core systems, bus-based SMB systems, NUMA-based SMP systems and taking advantage of processor affinity. It also includes error detection on mem-to-mem data transfer, using 32bit CRC on the I/O bus. Added latest is the network level error-detection mechanism over InfiniBand.

In the original design of MVAPICH intra-node communication utilized shared memory. Each pair of processes on the same node allocated two shared memory buffers between them for exchanging messages to each other exclusively. The memory that needed to be allocated for this was $P * (P-1) * BufSize$ where P is the number of processes and BufSize is the size of each shared buffer. Message ordering is ensured using the memory consistency model, and memory barrier if the underlying memory model is not consistent.

2.6.1 Improvements for Intra-node MPI communication

Chai et al.[8] designed and implemented a high performance and scalable MPI intra-node communication scheme, as an improvement of MVAPICH. Although their work concentrated on clusters, the main focus was on CMP, making it (near) ideal for this report. At the time of writing, they found that few studies had been done to study interaction between multi-core systems and MPI implementation. They designed a *user space memory copy* based architecture to improve intra-node communication. The goals included reducing latency, expanding bandwidth, and reduce memory usage for scalability. ((The work was later included in the official MVAPICH releases))

To avoid the use of locks, they separated buffers for small and large messages, and used a shared buffer pool for each process to send large messages. In their design each process had $P-1$ small *Receive Buffers*, one *Send Buffer Pool* and a collection of $P-1$ *Send Queues*. The sizes of the two latter can be tuned, Panda et al. used receive buffer size 32KB, one buffer cell to 8KB, and total number of cells to 128 for each send buffer pool. Small messages are simply written directly to the receiving process's receive buffer, which the receiving process then moves to its final spot.

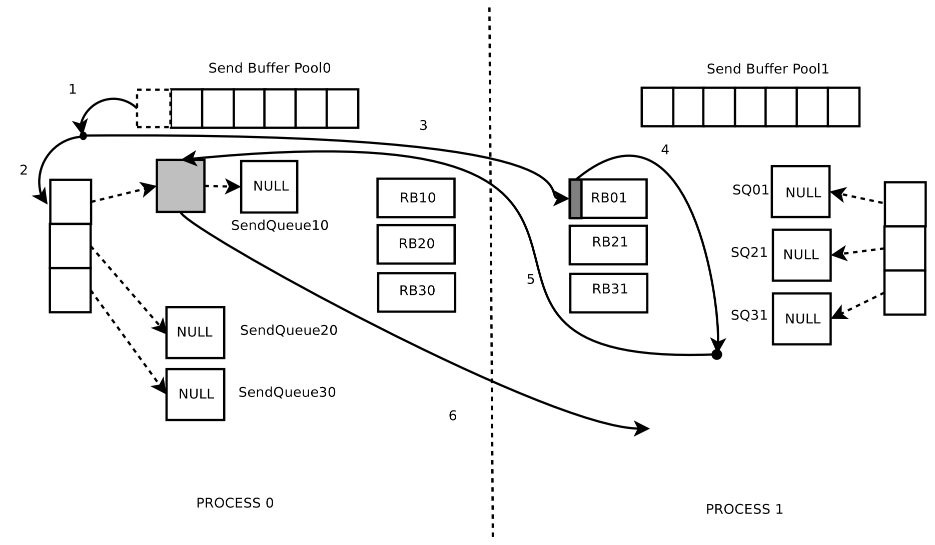

Figure 2.3: Mechanism for sending/receiving large messages

Transferring large messages are slightly more complicated(illustrated in figure 2.3), where the sending process puts the message in a free cell(1 & 2) of the send buffer pool, and then transmits a control message containing the address(3). The receiver reads(4) and accesses the address specified by the control message(5) and retrieves the message(6). The next time the send buffer pool is used again, the sender marks the send buffer pool-cell used free, a scheme called *mark-and-sweep* by Panda et al. When the message is bigger than one cell, it simply splits the message and transfers the cells individually.

Results On their Non-Uniform Memory Access(NUMA) cluster, composed of four nodes of AMD Opteron dual-core processors with 1MB L2 cache per core, they ran Linux 2.6.16. They achieved improved latency by up to 35% for large messages, small and medium up to 15%. Bandwidth was improved for most large messages about 50%. They also found that L2 cache miss rate on all benchmarks was improved drastically with their algorithm.

2.7 Understanding the impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System

Chai et al.[7] designed a set of experiments to study the impact of multi-core architecture on cluster computing. Their study included a cluster of 4 Intel Bensley systems connected by InfiniBand, each node having two dual-core 2.6GHz Woodcrest processors. They found that on average 50% of messages are transferred through intra-node communication. With data tiling, their benchmarking showed execution time improvements by up to 70%. However, their experiments showed that it became less efficient with more processes pr chip, and in experiments that spanned more processor-chips. With their setup, medium sized messages in the range 4KB to 64KB, they found that 10% of the messages was transferred intra-CMP, 30% was transferred through inter-CMP, and 60% was transferred through inter-node in the NAMD benchmark. According to their paper, the statistics show that if each core was supposed to communicate evenly with each other core, these numbers should have been 6.7%, 13.7% and 80% respectively. This indicates that the importance of optimizing intra-node communication is quite to the level of that of inter-node communication for this type of benchmark.

2.8 LAM/MPI

LAM/MPI[30, 6] is a software parallel software environment based on the Message-Passing Interface. LAM stands for Local Area Multicomputer and existed before the MPI-standard was established. LAM/MPI primarily targeted cluster environments, implementing various topologies to allow hetero-type environments.

2.8.1 Collective operations

LAM/MPI has modules for implementing MPI collective routines on different environments, shared memory or SMP. Except MPI_ALLTOALLW and MPI_EXSCAN the main collective operations is either optimized for SMP-environments or already optimized in the standard implementation of LAM/MPI[22]. The SMP module determines the locality of processes to set up a dynamic structure to perform the collective operation. The communication is still layered on

MPI point-to-point communication, but the algorithms attempt to maximize the use of on-node communication before communicating with off-node processes.

LAM/MPI also has a module for shared memory. It is only available when the communicator spans one single node, and the communicator can successfully attach the shared memory region to their address space. It uses two disjoint regions, one for synchronization and one for message passing, the last one divided into segments based on parameters, default value 8.

However, LAM/MPI is now only maintenance code, and as their web page states, the main work is now being done on Open MPI.

2.9 Open MPI

Open MPI is according to [9] influenced by LAM/MPI, LA-MPI and FT-MPI. It is an MPI-2 implementation that from the start supported shared memory architectures, Myrinet, Quadrics, Infiniband and TCP/IP. It supports multiple network interfaces on each node, and the usage of fail-overs transparent to the application. They have centered their design process around the MPI Component Architecture(MCA), with MCA backbone, component framework to manage modules, and modules with different parts of the implementation. These modules are loaded, used and unloaded by the framework, on demand. Examples of component frameworks in Open MPI is Point-to-point Transport Layer(PTL), Collective Communication (COLL), Reduction Operations and Parallel I/O. Modules is loaded during MPI_INIT, where the corresponding framework lists all available modules and loads the required ones. The highest priority module is used for the task, even though there might be multiple modules capable of doing the same work. At the destruction of the communicator, the module is unloaded and resources freed.

Chapter 3

Model

3.1 Benchmarking

To benchmark these different implementations, we have based our work on that of [23] by Larsgård. His work include benchmarking the (then) newly acquired IBM eServer p575+, compared to the previous supercomputers operated at NTNU. Following his methodology suggested benchmarking suites include HPL, Pallas and SimpleBench[23]. As Pallas has since been acquired by Intel, this benchmark was not easily obtained, and we have concentrated our work around HPL and SimpleBench.

3.1.1 Methodology

To benchmark the performance of the different implementations we are using the simplest benchmarks possible, with the same settings. We are looking at intra-node performance, and have therefore chosen not to benchmark the full system. Our experiments is only done on one node with 8 processes, one for each core. In this respect, the results is not directly comparable to other benchmarks done with the same tool.

3.1.2 HPL

High Performance Linpack(HPL)[27] benchmark is the basis of the top 500-list, and is a measure of a system's floating point computing power. It solves a dense *n by n* system of linear equations. It is portable and has several parameters to tune the benchmark for any system. To obtain the optimized parameters for Stallo, we have used the methods from Larsgård's report[23]. The optimal parameters are given in table5.1.

3.1.3 SimpleBench

SimpleBench is a simple benchmark for parallel computers, written by Thorvald Natvig[33] in connection with the acquisition of Njord, NTNU's IBM eServer system, in 2006. The purpose of SimpleBench is to reveal potential "spikes" in the system, where response time is high or performance worse than the average. It consists of a series of single runs of basic MPI-calls, with timing on each MPI-process.

The source code for SimpleBench is attached in appendix A.1.

3.1.4 Averages

To test the full potential of intra-node performance, one needs to eliminate any external factors that may offset the benchmark. SimpleBench does several runs of each collective operation, but only once with the same size payload. To be sure that the benchmark is independent one needs to test many iterations of each collective operation with the same parameters. Also, we need to make sure that no data can be reused directly from cache, nor any other benefits gained from looping over the same operation. This is achieved by scrolling through random unrelated data before sends.

Loops of 1000, 5000 and 10000 iterations of the collective operation all-to-all and loops of 1000 reduce is chosen, as these are frequently used. Timing of all-to-all operations showed little or no deviance in test results for more iterations, and did not exhibit the need for these lengthy test for the reduce operation as well. For all-to-all the tests includes sending and receiving of 400x50 doubles, 1000x1000 doubles, and 8000x8000 doubles. For reduce, 1, 1000x1000 and 8000x8000. 8000x8000 is used to get some results that will show utilization of memory bandwidth.

The source code for these benchmarks is attached in appendix B.1 and C.1.

3.1.5 Accuracy

The different implementations had all different results from MPI_Wtick, which returns the resolution of MPI_Wtime, used for timing in much of this report. MPICH, MVAPICH and Open MPI reported resolutions of, $9.536743 * 10^{-7}$, $1 * 10^{-5}$and $1 * 10^{-6}$ respectively on our system.

Chapter 4

Usability

All but one implementations installed cleanly, namely LAM/MPI. With Open MPI based on among others LAM/MPI, and since it was no longer in development, time would decide that this implementation was left out of this report, as far as results goes. MPICH was compiled with the ch_shmem device, MVAPICH with the ch_smp device, and Open MPI automatically chose its MCA-module "*linux*" for processor affinity.

Chapter 5

Results

5.1 Stallo

The system[13] used for our benchmarks consists of 704 dual-processor Xeon 2 nodes, clocked at 2.6GHz. Each node has a total of eight cores, 16GBytes memory and 120GBytes disk. A little over half the nodes is interconnected by InfiniBand(55%), and the rest has Gigabit Ethernet interconnect. Theoretical peak performance for the complete system is calculated to 60TFlops. The operating system used is Rock Linux[29].

5.2 HPL

The High-Performance Linpack benchmark results(table 5.2) shows that the different implementations performs almost on level. Open MPI performs only 0,38% better than MVAPICH and approximately 1,38% faster than MPICH. All the implementations had their best performance from the same set of parameters as given in table 5.1. Individually for MVAPICH and Open MPI, the different panel factorization solutions was giving results with little or no deviance from the average(<50MFlops difference), with the exception of the first test in each set. MPICH showed up to 5GFlops difference between best and worst panel factorization method.

Table 5.1: HPL parameters

	NB	N	PxQ
MPICH	256	44700	4x2
MVAPICH	256	44700	4x2
Open MPI	256	44700	4x2

Table 5.2: High-Performance Linpack Results

Implementation	HPL R_{max}
MPICH	4.136e+01
MVAPICH	4.177e+01
Open MPI	4.193e+01

Table 5.3: Minimum results from SimpleBench

Min				
Operation	Size	MPICH	MVAPICH	Open MPI
PingPong	0	22.888184	2.000000	21.219254
PingPong	100	31.948090	3.000000	10.967255
PingPong	1024	41.961670	22.000000	38.146973
PingPong	1M	22424.936295	16668.000000	14268.159866

5.3 SimpleBench

The MVAPICH results is clearly of lower resolution (MVAPICH returns resolution of $1.0 * 10^{-5}$ on this system (as opposed to $1.0 * 10^{-6}$for Open MPI)) than the rest of the implementations. From table 5.3 one can observe that MVAPICH has up to ten times shorter minimum send-times for small and medium sizes. At 1MB Open MPI is 15% faster than MVAPICH. MPICH is consistently the slowest implementation for minimum-times. The maximum-results from table 5.4 shows MVAPICH again faster for 0, 100 and 1024 bytes. Table 5.5 shows that MVAPICH generally outperforms both MPICH and Open MPI for 0, 100 and 1024 bytes, but falls short for Open MPI on 1MB. The average sending time for 0 bytes is over six times faster than for Open MPI.

5.4 Own benchmarking

The results from the iteration-intensive benchmarks shows further difference between the implementations. From table 5.6 and figure 5.1 we can clearly see that MPICH is performing far worse than the other implementations, especially with regards to minimum and maximum values. For small messages MPICH is

Table 5.4: Maximum results from SimpleBench

Maximum				
Operation	Size	MPICH	MVAPICH	Open MPI
PingPong	0	46.014786	16.000000	208.854675
PingPong	100	40.054321	9.000000	15.974045
PingPong	1024	64.849854	29.000000	93.936920
PingPong	1M	27837.991714	21351.0000000	19603.967667

Table 5.5: Average results SimpleBench

Average				
Operation	Size	MPICH	MVAPICH	Open MPI
PingPong	0	28.878450	8.2500000	53.226948
PingPong	100	35.643578	4.875000	11.742115
PingPong	1024	44.971704	23.500000	48.011541
PingPong	1M	21393.865347	16204.125000	15465.885401

Table 5.6: All-to-all, small sized messages(400x50 doubles)

Impl	Iter	Tot	Avg	Max	Min
MPICH	1000	8.176477E-01	8.176477E-04	1.471901E-02	6.039143E-04
MPICH	5000	4.013466E+00	8.026933E-04	4.582882E-03	4.930496E-04
MPICH	10000	8.080390E+00	8.080390E-04	1.489401E-02	3.039837E-04
MVAPICH	1000	5.766840E-01	5.766840E-04	1.302000E-03	3.720000E-04
MVAPICH	5000	2.846303E+00	5.692606E-04	1.235900E-02	3.580000E-04
MVAPICH	10000	5.784503E+00	5.784503E-04	2.397600E-02	3.630000E-04
Open MPI	1000	5.006273E-01	5.006273E-04	8.707047E-03	4.441738E-04
Open MPI	5000	2.457770E+00	4.915540E-04	4.745388E-02	4.379749E-04
Open MPI	10000	4.768914E+00	4.768914E-04	6.028891E-03	4.169941E-04

up to 35% slower than MVAPICH and up to 45% slower than Open MPI. Both with regards to minimum and maximum values for small messages, MVAPICH has the shortest times. But with average and total times, we can see that Open MPI is actually outperforming MVAPICH by almost 20% for the test-run of 10000 iterations. For medium sized messages, as seen in figure 5.1 and figure 5.4(without MPICH), the pattern repeats itself. The distance in time up to MPICH is growing fast, but the performance of MVAPICH and Open MPI is evening out. Open MPI still has the upper hand on averages and MVAPICH on the best minimum and maximum times. For large messages(figure 5.5 and table 5.8), Open MPI is best by all measures, by as much as over two tenth of a second per iteration from best to worst average result.

For the reduce-operation benchmark, we observe almost the same behavior, except that here MVAPICH is outperforming Open MPI on the average results for the smallest data set. The medium data set is showing best minimum time to MVAPICH, but Open MPI average is much better.

Table 5.7: All-to-all, medium sized messages(1000x1000 doubles)

Impl	Iter	Tot	Avg	Max	Min
MPICH	1000	8.543125e+02	8.543125e-01	2.437500e+00	1.445312e-01
MPICH	5000	5.161012e+03	1.032202e+00	2.808594e+00	1.679688e-01
MPICH	10000	8.793560e+03	8.793560e-01	1.679313e+01	4.113698e-02
MVAPICH	1000	4.820071e+01	4.820071e-02	7.076800e-02	3.094600e-02
MVAPICH	5000	2.461737e+02	4.923474e-02	7.342500e-02	3.061900e-02
MVAPICH	10000	4.914969e+02	4.914969e-02	7.468300e-02	3.080700e-02
Open MPI	1000	4.704773e+01	4.704773e-02	6.683207e-02	4.432511e-02
Open MPI	5000	2.317214e+02	4.634429e-02	7.062197e-02	3.633189e-02
Open MPI	10000	4.616412e+02	4.616412e-02	6.941700e-02	3.950596e-02

Table 5.8: All-to-all, large sized messages(8000x8000 doubles)

Impl	Iter	Tot	Avg	Max	Min
MVAPICH	1000	3.181639e+03	3.181638e+00	4.227398e+00	3.070103e+00
MVAPICH	5000	1.600605e+04	3.201210e+00	5.382317e+00	3.050804e+00
Open MPI	1000	3.039166e+03	3.039166e+00	3.960890e+00	2.772568e+00
Open MPI	5000	1.352857e+04	2.705714e+00	3.590074e+00	2.606230e+00

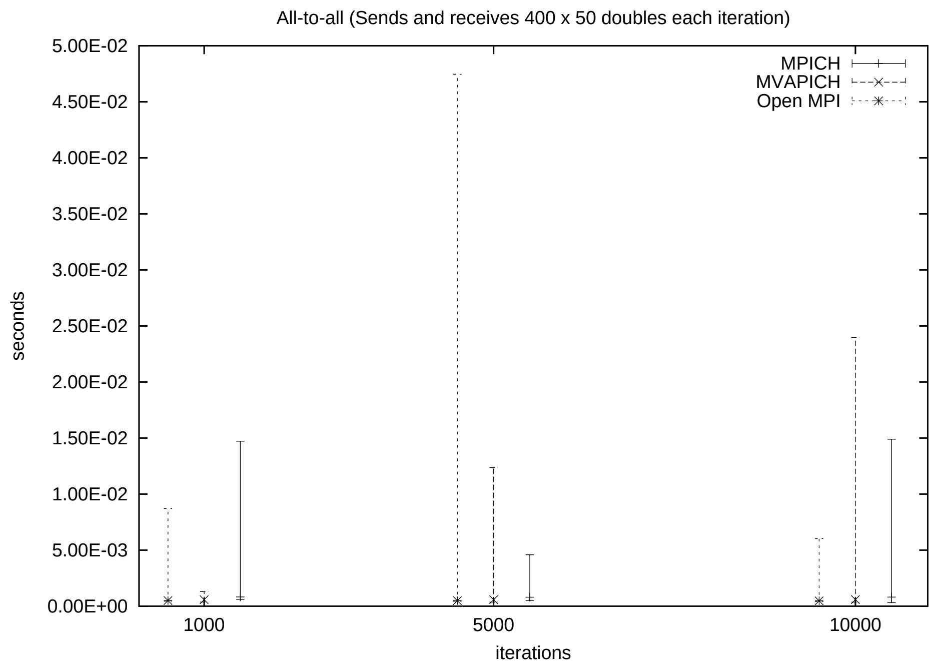

Figure 5.1: All-to-all, MPICH, MVAPICH, Open MPI

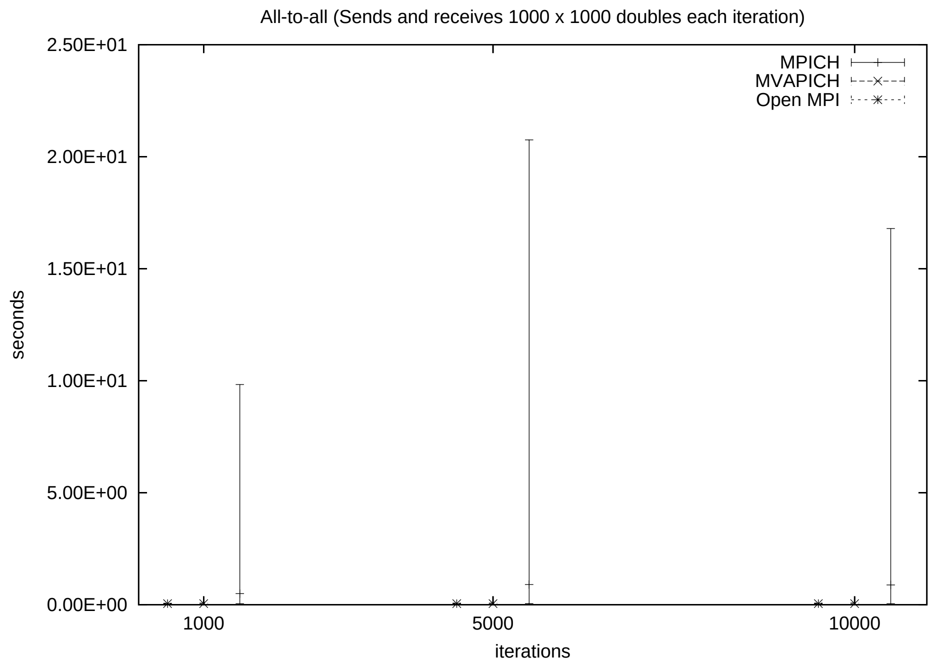

Figure 5.2: All-to-all, medium size messages, MPICH, MVAPICH, Open MPI

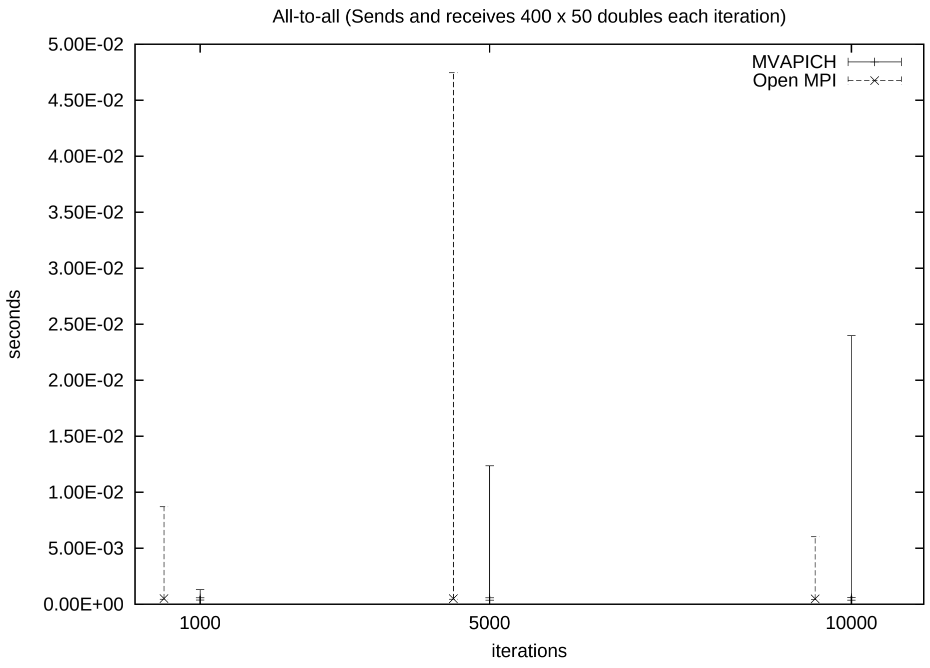

Figure 5.3: All-to-all, small messages, MVAPICH, Open MPI

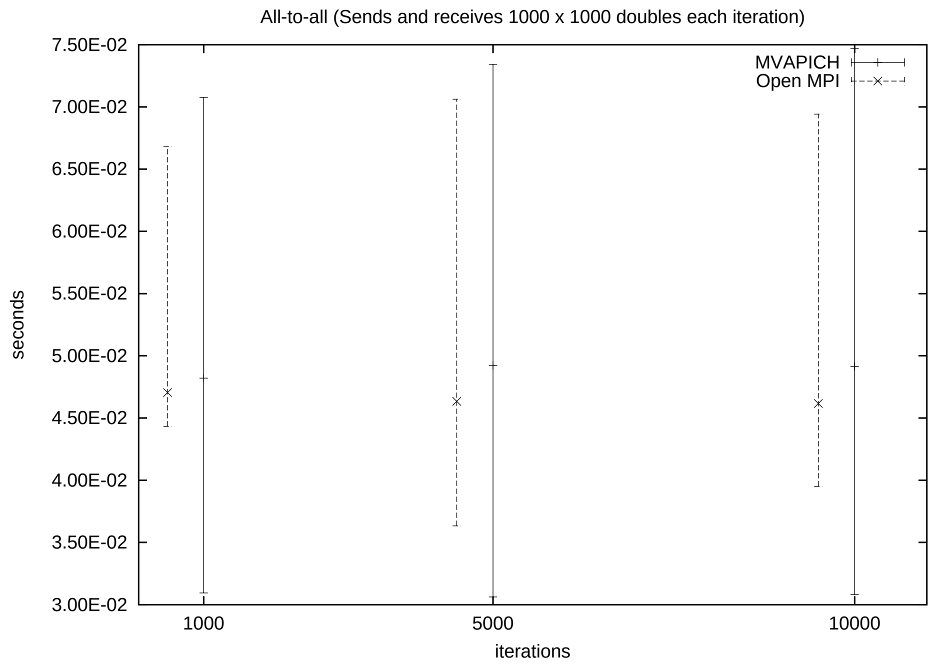

Figure 5.4: All-to-all, medium sized messages, MVAPICH, Open MPI

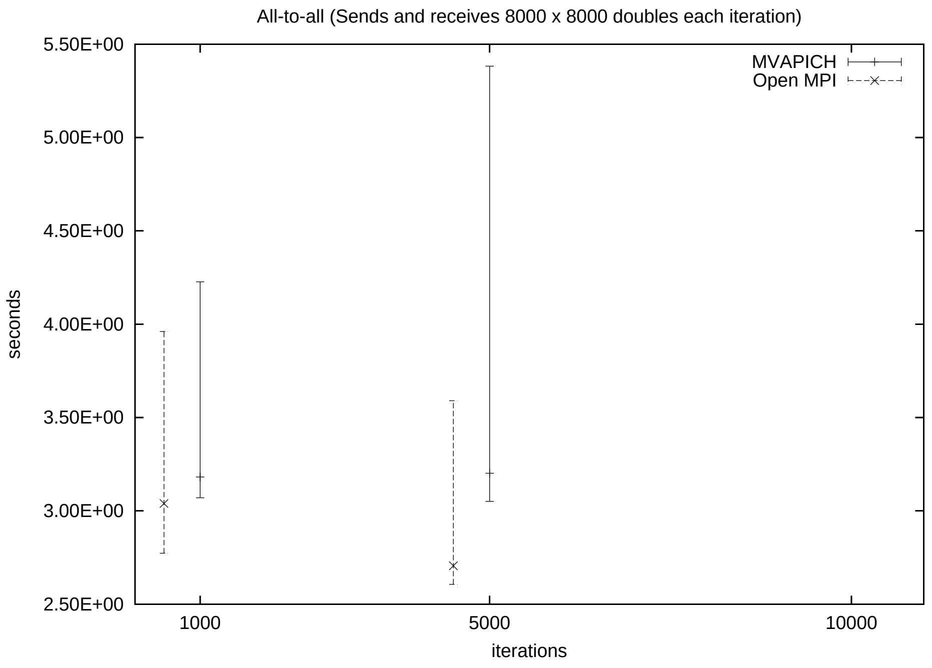

Figure 5.5: All-to-all, large messages, MVAPICH, Open MPI

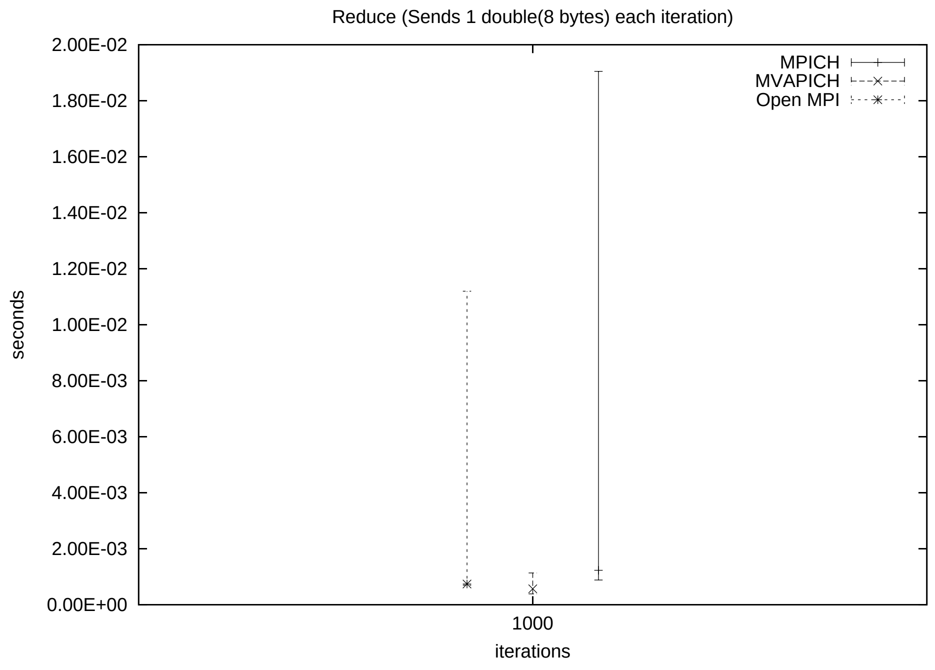

Figure 5.6: Reduce operation with 1 double each run

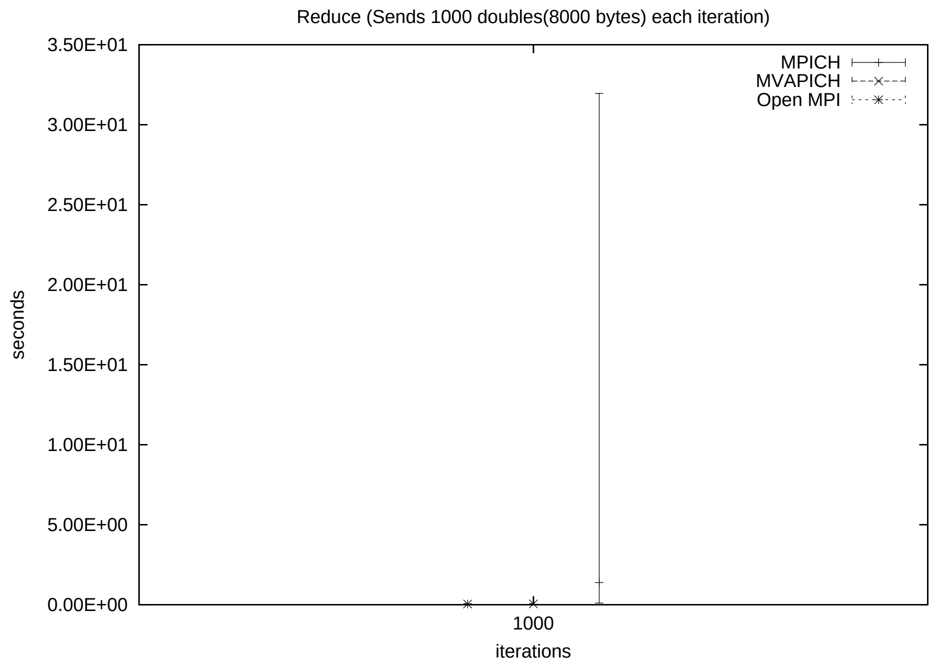

Figure 5.7: Reduce operation with 1000 doubles each run

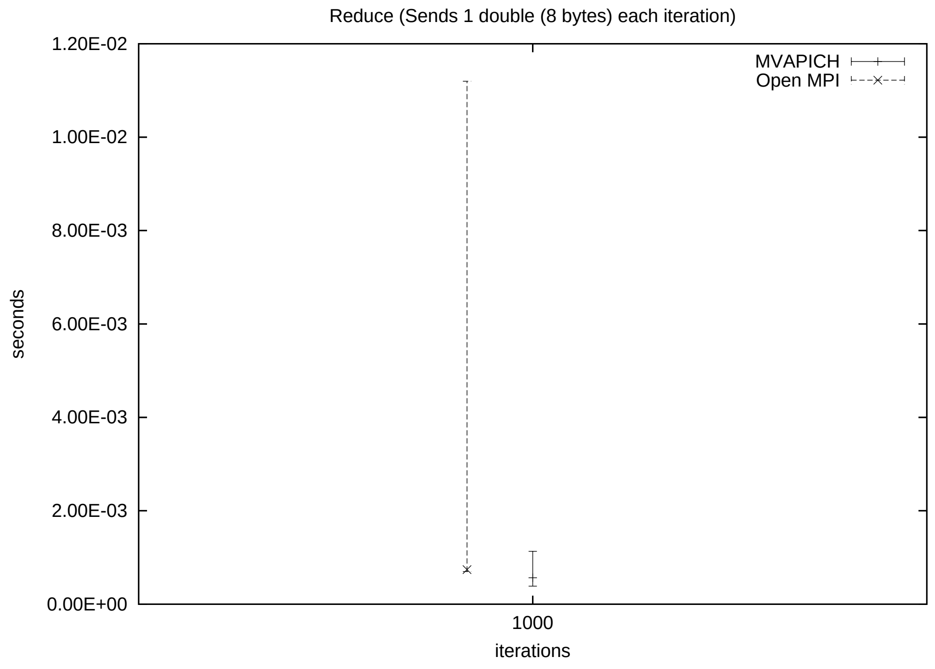

Figure 5.8: Reduce operation with 1 double each run(MPICH excluded)

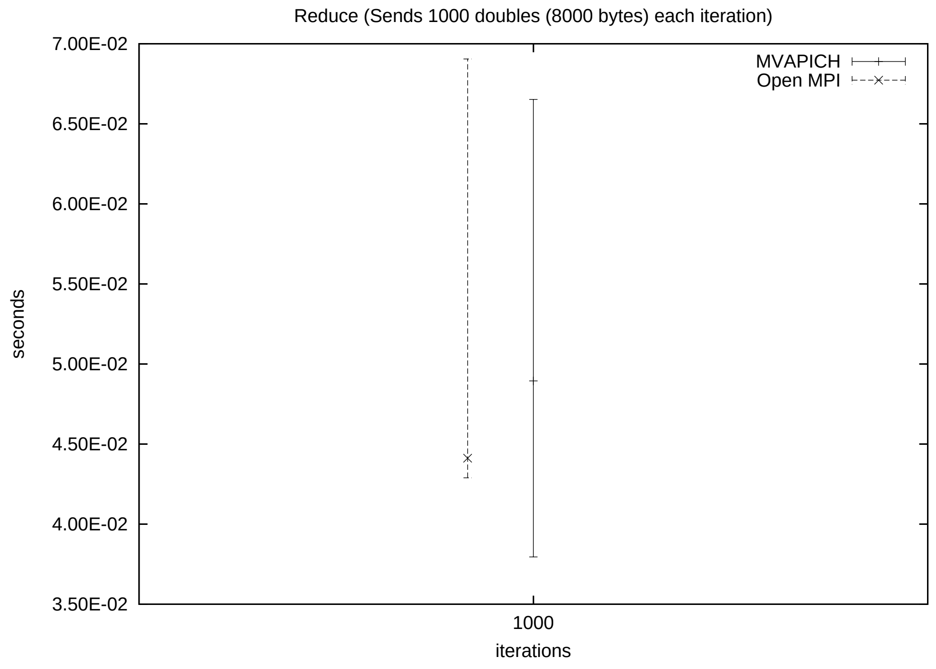

Figure 5.9: Reduce operation with 1000 doubles each (MPICH excluded)

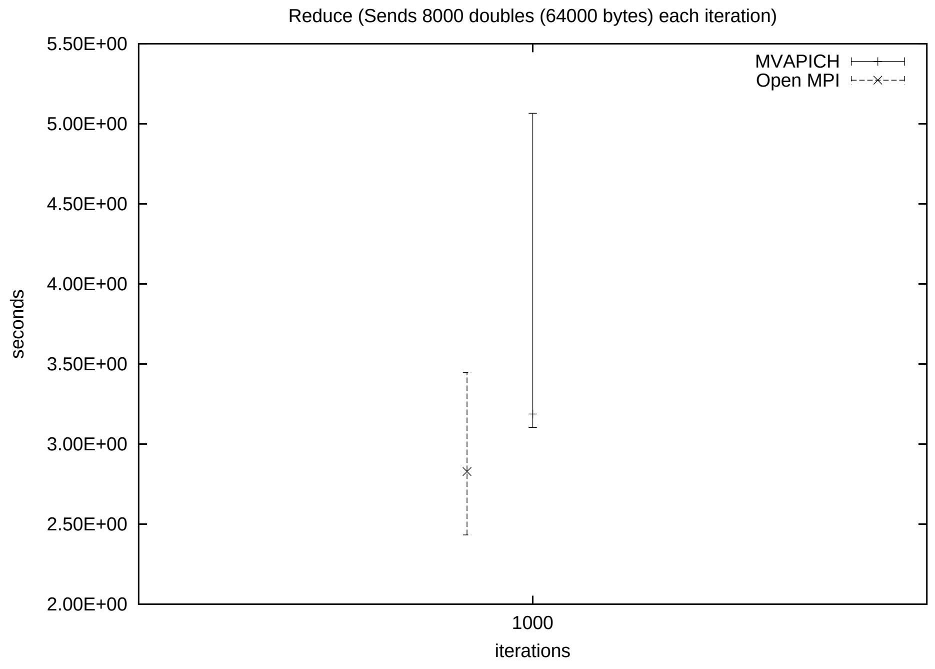

Figure 5.10: Reduce operation with 8000 doubles each run(MPICH excluded)

Chapter 6

Conclusions

MVAPICH outperformed Open MPI for small and medium sized messages in the all-to-all-department, on minimum time spent on sending and receiving. However, the average time, which stays quite level for Open MPI and MVAPICH when iteration is scaled, was better for Open MPI. The first iteration of each run was mainly the slowest one, and this offsets the total. But when we examined the results with the first and last five iterations excluded, MVAPICH was still behind on the average results. Since MVAPICH was both faster on the minimum and slower on the maximum, one can only presume that it might have something to do with the accuracy of the implementation. Although, even if that was the case, we did not approaching tick-time for neither MVAPICH nor Open MPI.

The results from the HPL benchmark showed us that Open MPI is the fastest implementation for solving a "*real life-problem*" that one might expect the supercomputer to evaluate. The results differed by less than 2%, so it is not easy to be very conclusive on the basis of this test.

Overall, we have seen Open MPI outperform both MPICH and MVAPICH on average results and the HPL benchmark. MVAPICH did the fastest iterations for small data sets, but came up short when bandwidth became an issue.

Chapter 7

Future Work

This report has shown that MVAPICH is better for small data sets. To reveal the limits of the upper hand of MVAPICH, a more detailed benchmark is required, both in terms of sizes of data sets, and in terms of different calls and algorithms. A natural next step is to scale up and try different data- and processor-layout schemes, to benchmark performance across interconnects.

Bibliography

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Dough Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.

[2] AMD Athlon Dual-Core Product Brief. Available at: http://www.amd.com/us-en/Processors/ProductInformation/ 0,,30_118_9485_13041î3078,00.html.

[3] AMD Opteron Quad-Core Product Brief. Available at: http://www.amd.com/us-en/Processors/ProductInformation/ 0,,30_118_8796_15223,00.html.

[4] Michael Barnett, Satya Gupta, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Interprocessor Collective Communications Library (Intercom). In *Proceedings of the Scalable High Performance Computing Conference*, pages 357–364. IEEE Computer Society Press, 1994.

[5] Douglas C. Bossen, Joel M. Tendler, and Kevin Reick. Power4 system design for high reliability. *IEEE Micro*, 22(2):16–24, 2002.

[6] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[7] Lei Chai, Qi Gao, and Dhabaleswar K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *CCGRID*, pages 471–478. IEEE Computer Society, 2007.

[8] Lei Chai, Albert Hartono, and Dhabaleswar K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *CLUSTER*. IEEE, 2006.

[9] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[11] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann, September 2006.

[13] The High Perfomance Computing at the University of Tromsø. Webpage: http://docs.notur.no/uit/.

[14] J. Huh, D. Burger, and S. Keckler. Exploring the design space of future CMPs, 2001.

[15] IBM System p5 Quad-Core Module Based on POWER5+ Technology: Techincal Overview and Introduction. Available at: http://www.redbooks.ibm.com/redpapers/pdfs/redp4150.pdf.

[16] Intel Architecture and Silicone Cadence, The Catalyst for Industry Innovation. Available at: http://download.intel.com/technology/eep/cadence-paper.pdf.

[17] Quad-Core Intel® Xeon® Processor 5300 Series Product Brief (Clowertown). Available at: http://download.intel.com/products/processor/xeon/dc53kprodbrief.pdf.

[18] Dual-Core Intel Xeon Processor Paxville. Available at: ftp://download.intel.com/design/Xeon/datashts/30915801.pdf (Retrieved: 30.01.2008).

[19] Dual-Core Intel Xeon Processor 7100 Series Product Brief (Tulsa). Available at: http://download.intel.com/products/processor/xeon/7100_prodbrief.pdf.

[20] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[21] L.A.Barroso, K.Gharachoroloo, R.McNamara, Andreas Nowatzyk, S.Qadeer, B.Sano, S.Smith, R.Stets, and B.Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[22] LAM/MPI User's Guide. Available at: http://www.lam-mpi.org/download/files/7.1.4-user.pdf.

[23] Nils Magnus Larsgård. Benchmarking of Modern Supercomputers. December 2006.

[24] Message-Passing Interface Forum. http://www.mpi-forum.org.

[25] MVAPICH - MPI over InfiniBand. Webpage: http://mvapich.cse.ohio-state.edu/.

[26] MVICH - MPI for Virtual Interface Architecture. Webpage: http://crd.lbl.gov/FTG/MVICH/mvich.shtml.

[27] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. High-Performance Linpack Benchmark for Distributed-Memory Computers. Website: http://www.netlib.org/benchmark/hpl/index.html.

[28] Rolf Rabenseifner. A new optimized MPI reduce algorithm. November 1997.

[29] Rock Linux. Official webpage: http://www.rocklinux.org.

[30] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[31] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH, October 2003.

[32] Rajeev Thakur and William Gropp. Open Issues in MPI Implementation. In Lynn Choi, Yunheung Paek, and Sangyeun Cho, editors, *Asia-Pacific Computer Systems Architecture Conference*, volume 4697 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2007.

[33] Personal communication with Thorvald Natvig, PhD Candidate NTNU.

[34] M.K. Velamati, A. Kumar, N. Jayam, G. Senthilkumar, P.K. Baruah, S. Kapoor, R. Sharma, and A. Srinivasan. Optimization of Collective Communication in Intra-Cell MPI. In *14th IEEE International Conference on High Performance Computing (HiPC)*, pages 488–499, 2007.

[35] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

Appendix A

SimpleBench

A.1 Code

SimpleBench by Thorvald Natvig[33]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"
#include <math.h>
typedef double timetype;

// Support routines
static void timing_start(timetype *t);
static void timing_stop(timetype *t);
static double elapsed_us(timetype *start, timetype *stop);
static void minmaxelapse(timetype *start, timetype *stop, double *min,
                         double *max);
static void expand_benchm_lists(int **bl);

// Benchmark routines
void pingpong(int size);
void alltoall(int size);
void collective(int size);
void cartesian(int size);

// Benchmark configuration
#define SB_EOL   -1
#define SB_NODES -2
int pingpong_sizes[] = { 0, 100, 1024, 8192, 1000000, SB_EOL };
int alltoall_sizes[] = { SB_NODES, 8192, 100000, SB_EOL };
int collective_sizes[] = { 0, SB_NODES, 8192, 1000000, SB_EOL };
int cartesian_sizes[] = { SB_NODES, 10000, 20000, 40000, SB_EOL };
/* removed 80 000 and 160 000 */
int *benchm_lists[] = { pingpong_sizes, alltoall_sizes, collective_sizes,
                        cartesian_sizes, NULL };


int rank;
int nodes;

int
main(int argc, char **argv) {
  int *sz;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  expand_benchm_lists(benchm_lists);

  if (rank == 0)
    printf("Running on %d nodes\n", nodes);
  // Dryrun once
  pingpong(0);

  for (sz = pingpong_sizes; *sz != SB_EOL; sz++)
    pingpong(*sz);

  for (sz = alltoall_sizes; *sz != SB_EOL; sz++)
    alltoall(*sz);

  for (sz = collective_sizes; *sz != SB_EOL; sz++)
    collective(*sz);

  for (sz = cartesian_sizes; *sz != SB_EOL; sz++)
    cartesian(*sz);

  MPI_Finalize();
  return 0;
}

// Support routines

static void
timing_start(timetype *t) {
  *t = MPI_Wtime();
}

static void
timing_stop(timetype *t) {
  *t = MPI_Wtime();
}

static double
elapsed_us(timetype *start, timetype *stop) {
  return (*stop - *start) * 1000000;
}

static void
minmaxelapse(timetype *start, timetype *stop, double *min, double *max) {
  double elapsed = elapsed_us(start, stop);
  MPI_Allreduce(&elapsed, min, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
  MPI_Allreduce(&elapsed, max, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
}

static void
expand_benchm_lists(int **bl) {
  int *sz;
  for (; *bl; bl++)
    for (sz = *bl; *sz != SB_EOL; sz++)
      if (*sz == SB_NODES)
        *sz = nodes;
}

// Simple pingpong tests to test minimum and maximum
// latency and bandwidth.

void
pingpong(int size) {
  MPI_Status stat;
  timetype start, stop;
  double min, max, elapsed, total;
  double *buf;
  int i;
  min = 1000000000.0;
  max = 0.0;
  total = 0.0;

  buf = (double *) malloc(size * sizeof(double));

  MPI_Barrier(MPI_COMM_WORLD);

  if (rank == 0) {
    for(i=0;i<size;i++)
      buf[i]=(i*7);
    for(i=1;i<nodes;i++) {
      timing_start(&start);
      MPI_Send(buf, size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      MPI_Recv(buf, size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &stat);
      timing_stop(&stop);
      elapsed = elapsed_us(&start, &stop);
      if (elapsed < min)
        min = elapsed;
      if (elapsed > max)
        max = elapsed;
      total += elapsed;
    }
    printf("Pingpong (%d doubles, %u bytes): %f -> %f [%f]\n",size,
           size * sizeof(double),min,max,total / (double)nodes);

  } else {
    MPI_Recv(buf, size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);
    MPI_Send(buf, size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  }

  free(buf);
}


// Tests of alltoall

void
alltoall(int size) {
  timetype start, stop;
  double min, max;
  double *sendbuf, *recvbuf;
  int i;
  int localsize;
  MPI_Comm halfcomm;

  localsize = size / nodes;
  sendbuf = (double *) malloc(sizeof(double) * size * 2);
  recvbuf = (double *) malloc(sizeof(double) * size * 2);
  if (sendbuf==NULL || recvbuf==NULL)
      perror("alltoall");

  for(i=0;i<localsize*nodes;i++)
    sendbuf[i]=i*rank;

  MPI_Barrier(MPI_COMM_WORLD);
  timing_start(&start);
  MPI_Alltoall(sendbuf, localsize, MPI_DOUBLE, recvbuf, localsize,
               MPI_DOUBLE, MPI_COMM_WORLD);
  timing_stop(&stop);

  minmaxelapse(&start, &stop, &min, &max);

  if (rank == 0) {
    printf("Alltoall (%d doubles/%u bytes), %f -> %f\n", localsize * nodes,
           localsize * nodes * sizeof(double), min, max);
  }
  MPI_Comm_split(MPI_COMM_WORLD, (rank*2 >= nodes) ? 1 : 0, rank, &halfcomm);
  MPI_Barrier(MPI_COMM_WORLD);
  timing_start(&start);
  MPI_Alltoall(sendbuf, localsize * 2, MPI_DOUBLE, recvbuf, localsize * 2,
               MPI_DOUBLE, halfcomm);
  timing_stop(&stop);
  MPI_Comm_free(&halfcomm);

  minmaxelapse(&start, &stop, &min, &max);

  if (rank == 0) {
    printf("Alltoall (%d doubles/%u bytes), %f -> %f [Split Halves]\n",
           localsize * nodes, localsize * nodes * sizeof(double), min, max);
  }

  free(sendbuf);
  free(recvbuf);
}


// Collective operations

void
collective(int size) {
  timetype start, stop;
  double min, max;
  double *sendbuf, *recvbuf;
  int i;
  int localsize;
  MPI_Comm halfcomm;

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Comm_split(MPI_COMM_WORLD, (rank*2 >= nodes) ? 1 : 0, rank, &halfcomm);

  if (size == 0) {
    timing_start(&start);
    MPI_Barrier(MPI_COMM_WORLD);
    timing_stop(&stop);

    minmaxelapse(&start, &stop, &min, &max);
    if (rank == 0)
      printf("Barrier in %f -> %f\n", min, max);

    timing_start(&start);
    MPI_Barrier(halfcomm);
    timing_stop(&stop);

    minmaxelapse(&start, &stop, &min, &max);
    if (rank == 0)
      printf("Barrier in %f -> %f [Halfcomm]\n", min, max);
  }

  localsize = size / nodes;
  sendbuf = (double *) malloc(sizeof(double) * localsize);
  recvbuf = (double *) malloc(sizeof(double) * localsize);

  for(i=0;i<localsize;i++)
    sendbuf[i] = rank * i;

  MPI_Barrier(MPI_COMM_WORLD);

  timing_start(&start);
  MPI_Reduce(sendbuf, recvbuf, localsize, MPI_DOUBLE, MPI_MIN, 0,
             MPI_COMM_WORLD);
  timing_stop(&stop);

  minmaxelapse(&start, &stop, &min, &max);
  if (rank == 0)
    printf("Reduce    (min) (%d doubles, %u bytes) %f -> %f\n",
           localsize * nodes, localsize * nodes * sizeof(double), min, max);

  MPI_Barrier(MPI_COMM_WORLD);
  timing_start(&start);
  MPI_Allreduce(sendbuf, recvbuf, localsize, MPI_DOUBLE, MPI_MIN,
                MPI_COMM_WORLD);
  timing_stop(&stop);

  minmaxelapse(&start, &stop, &min, &max);
  if (rank == 0)
    printf("AllReduce (min) (%d doubles, %u bytes) %f -> %f\n",
           localsize * nodes, localsize * nodes * sizeof(double), min, max);

  MPI_Barrier(MPI_COMM_WORLD);
  timing_start(&start);
  MPI_Allreduce(sendbuf, recvbuf, localsize, MPI_DOUBLE, MPI_MIN, halfcomm);
  timing_stop(&stop);

  minmaxelapse(&start, &stop, &min, &max);
  if (rank == 0)
    printf("AllReduce (min) (%d doubles, %u bytes) %f -> %f [HalfComm]\n",
           localsize * nodes, localsize * nodes * sizeof(double), min, max);

  MPI_Barrier(MPI_COMM_WORLD);
  timing_start(&start);
  MPI_Allreduce(sendbuf, recvbuf, localsize, MPI_DOUBLE, MPI_SUM, halfcomm);
  timing_stop(&stop);

  minmaxelapse(&start, &stop, &min, &max);
  if (rank == 0)
    printf("AllReduce (add) (%d doubles, %u bytes) %f -> %f [HalfComm]\n",
           localsize * nodes, localsize * nodes * sizeof(double), min, max);

  free(sendbuf);
  free(recvbuf);
  MPI_Comm_free(&halfcomm);
}


// 2D Cartesian border exchange with datatypes

void
cartesian(int size) {
  int width, height;
  int left, right, above, below;
  int dims[2];
  int periods[2];
  MPI_Comm cartcomm;
  timetype start, stop;
  double min, max;
  int i;
  double *subarea;
  double *areastart;
  double *upperleft;
  double *upperright;
  double *lowerleft;
  double *rowabove;
  double *rowbelow;
  double *columnleft;
  double *columnright;
  MPI_Datatype column;
  MPI_Status stat;
  double elapsed[4];
  double emin[4], emax[4];
  MPI_Request req[8];
  MPI_Status rstat[8];
```

```c
periods[0] = periods[1] = 0;
dims[0] = dims[1] = 0;

MPI_Barrier(MPI_COMM_WORLD);

timing_start(&start);
MPI_Dims_create(nodes, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &cartcomm);
timing_stop(&stop);

width = size / dims[0];
height = size / dims[1];
minmaxelapse(&start, &stop, &min, &max);
if (rank == 0)
  printf("\nCartesian creation [%d] %f -> %f [%d x %d]\n", size, min, max,
         width, height);

MPI_Cart_shift(cartcomm, 0, 1, &left, &right);
MPI_Cart_shift(cartcomm, 1, 1, &above, &below);

subarea = malloc((width + 2) * (height + 2) * sizeof(double));
if(subarea == NULL) perror("malloc returned null");
areastart = subarea + width + 3;
upperleft = areastart;
upperright = areastart + width - 1;
lowerleft = areastart + (width + 2) * (height - 1);

rowabove = subarea + 1;
rowbelow = subarea + (width + 2) * (height + 1) + 1;
columnleft = subarea + width + 2;
columnright = subarea + width - 1;

MPI_Barrier(MPI_COMM_WORLD);
timing_start(&start);
MPI_Type_vector(height, 1, width + 2, MPI_DOUBLE, &column);
MPI_Type_commit(&column);
timing_stop(&stop);

minmaxelapse(&start, &stop, &min, &max);
if (rank == 0)
  printf("Cartesian datatype %f -> %f\n", min, max);

MPI_Barrier(MPI_COMM_WORLD);
for(i=0;i<4;i++) {
  timing_start(&start);
  MPI_Sendrecv(upperleft, 1, column, left, 0, columnright, 1, column,
               right, 0, cartcomm, &stat);
  MPI_Sendrecv(upperright, 1, column, right, 0, columnleft, 1, column,
               left, 0,  cartcomm, &stat);

  MPI_Sendrecv(upperleft, width, MPI_DOUBLE, above, 0, rowbelow, width,
               MPI_DOUBLE, below, 0, cartcomm, &stat);
  MPI_Sendrecv(lowerleft, width, MPI_DOUBLE, below, 0, rowabove, width,
               MPI_DOUBLE, above, 0, cartcomm, &stat);
  timing_stop(&stop);
  elapsed[i] = elapsed_us(&start, &stop);
}

MPI_Reduce(elapsed, emin, 4, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
MPI_Reduce(elapsed, emax, 4, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

if (rank == 0)
  for(i=0;i<4;i++)
    printf("Cartesian sendrecv iteration %d: %f -> %f\n", i, emin[i],
             emax[i]);

MPI_Barrier(MPI_COMM_WORLD);
```

```
  for(i=0;i<4;i++) {
    timing_start(&start);
    MPI_Irecv(columnright, 1, column, right, 0, cartcomm, &req[0]);
    MPI_Irecv(columnleft, 1, column, left, 0, cartcomm, &req[1]);
    MPI_Irecv(rowbelow, width, MPI_DOUBLE, below, 0, cartcomm, &req[2]);
    MPI_Irecv(rowabove, width, MPI_DOUBLE, above, 0, cartcomm, &req[3]);


    MPI_Isend(upperleft, 1, column, left, 0, cartcomm, &req[4]);
    MPI_Isend(upperright, 1, column, right, 0, cartcomm, &req[5]);
    MPI_Isend(upperleft, width, MPI_DOUBLE, above, 0, cartcomm, &req[6]);
    MPI_Isend(lowerleft, width, MPI_DOUBLE, below, 0, cartcomm, &req[7]);
    MPI_Waitall(8, req, rstat);
    timing_stop(&stop);
    elapsed[i]=elapsed_us(&start, &stop);
  }
  MPI_Reduce(elapsed, emin, 4, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
  MPI_Reduce(elapsed, emax, 4, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

  if (rank == 0)
    for(i=0;i<4;i++)
      printf("Cartesian isend/irecv iteration %d: %f -> %f\n", i,
             emin[i], emax[i]);
  MPI_Barrier(MPI_COMM_WORLD);

  free(subarea);
  MPI_Comm_free(&cartcomm);
}
```

A.2 Results

A.2.1 MPICH

```
Running on 8 nodes
Pingpong (0 doubles, 0 bytes): 0.000000 -> 0.000000 [0.000000]
Pingpong (0 doubles, 0 bytes): 0.000000 -> 3906.250000 [488.281250]
Pingpong (100 doubles, 800 bytes): 0.000000 -> 0.000000 [0.000000]
Pingpong (1024 doubles, 8192 bytes): 0.000000 -> 3906.250000 [488.281250]
Pingpong (8192 doubles, 65536 bytes): 0.000000 -> 3906.250000 [1953.125000]
Pingpong (1000000 doubles, 8000000 bytes): 183593.750000 -> 191406.250000 [164550.781250]
Alltoall (8 doubles/64 bytes), 0.000000 -> 7812.500000
Alltoall (8 doubles/64 bytes), 0.000000 -> 3906.250000 [Split Halves]
Alltoall (8192 doubles/65536 bytes), 0.000000 -> 3906.250000
Alltoall (8192 doubles/65536 bytes), 0.000000 -> 3906.250000 [Split Halves]
Alltoall (100000 doubles/800000 bytes), 15625.000000 -> 23437.500000
Alltoall (100000 doubles/800000 bytes), 7812.500000 -> 11718.750000 [Split Halves]
Barrier in 0.000000 -> 0.000000
Barrier in 0.000000 -> 3906.250000 [Halfcomm]
Reduce    (min) (0 doubles, 0 bytes) 0.000000 -> 0.000000
AllReduce (min) (0 doubles, 0 bytes) 0.000000 -> 0.000000
AllReduce (min) (0 doubles, 0 bytes) 0.000000 -> 0.000000 [HalfComm]
AllReduce (add) (0 doubles, 0 bytes) 0.000000 -> 0.000000 [HalfComm]
Reduce    (min) (8 doubles, 64 bytes) 0.000000 -> 0.000000
AllReduce (min) (8 doubles, 64 bytes) 0.000000 -> 0.000000
AllReduce (min) (8 doubles, 64 bytes) 0.000000 -> 0.000000 [HalfComm]
AllReduce (add) (8 doubles, 64 bytes) 0.000000 -> 0.000000 [HalfComm]
Reduce    (min) (8192 doubles, 65536 bytes) 0.000000 -> 0.000000
AllReduce (min) (8192 doubles, 65536 bytes) 0.000000 -> 3906.250000
AllReduce (min) (8192 doubles, 65536 bytes) 0.000000 -> 0.000000 [HalfComm]
AllReduce (add) (8192 doubles, 65536 bytes) 0.000000 -> 3906.250000 [HalfComm]
Reduce    (min) (1000000 doubles, 8000000 bytes) 19531.250000 -> 35156.250000
AllReduce (min) (1000000 doubles, 8000000 bytes) 23437.500000 -> 39062.500000
AllReduce (min) (1000000 doubles, 8000000 bytes) 15625.000000 -> 39062.500000 [HalfComm]
AllReduce (add) (1000000 doubles, 8000000 bytes) 19531.250000 -> 35156.250000 [HalfComm]
Cartesian creation [8] 0.000000 -> 3906.250000 [2 x 4]
Cartesian datatype 0.000000 -> 0.000000
Cartesian sendrecv iteration 0: 0.000000 -> 0.000000
Cartesian sendrecv iteration 1: 0.000000 -> 0.000000
Cartesian sendrecv iteration 2: 0.000000 -> 0.000000
Cartesian sendrecv iteration 3: 0.000000 -> 3906.250000
Cartesian isend/irecv iteration 0: 0.000000 -> 0.000000
Cartesian isend/irecv iteration 1: 0.000000 -> 0.000000
Cartesian isend/irecv iteration 2: 0.000000 -> 0.000000
Cartesian isend/irecv iteration 3: 0.000000 -> 0.000000
Cartesian creation [10000] 0.000000 -> 0.000000 [2500 x 5000]
Cartesian datatype 0.000000 -> 0.000000
Cartesian sendrecv iteration 0: 46875.000000 -> 82031.250000
Cartesian sendrecv iteration 1: 0.000000 -> 19531.250000
Cartesian sendrecv iteration 2: 0.000000 -> 19531.250000
Cartesian sendrecv iteration 3: 0.000000 -> 19531.250000
Cartesian isend/irecv iteration 0: 0.000000 -> 3906.250000
Cartesian isend/irecv iteration 1: 0.000000 -> 3906.250000
Cartesian isend/irecv iteration 2: 0.000000 -> 3906.250000
Cartesian isend/irecv iteration 3: 0.000000 -> 3906.250000
Cartesian creation [20000] 0.000000 -> 0.000000 [5000 x 10000]
Cartesian datatype 0.000000 -> 0.000000
Cartesian sendrecv iteration 0: 54687.500000 -> 128906.250000
Cartesian sendrecv iteration 1: 3906.250000 -> 46875.000000
Cartesian sendrecv iteration 2: 3906.250000 -> 7812.500000
Cartesian sendrecv iteration 3: 3906.250000 -> 15625.000000
Cartesian isend/irecv iteration 0: 3906.250000 -> 11718.750000
Cartesian isend/irecv iteration 1: 3906.250000 -> 11718.750000
Cartesian isend/irecv iteration 2: 3906.250000 -> 15625.000000
Cartesian isend/irecv iteration 3: 0.000000 -> 19531.250000
```

A.2.2 MVAPICH

```
Running on 8 nodes
Pingpong (0 doubles, 0 bytes): 2.000000 -> 16.000000 [8.250000]
Pingpong (0 doubles, 0 bytes): 2.000000 -> 4.000000 [2.625000]
Pingpong (100 doubles, 800 bytes): 3.000000 -> 9.000000 [4.875000]
Pingpong (1024 doubles, 8192 bytes): 22.000000 -> 29.000000 [23.500000]
Pingpong (8192 doubles, 65536 bytes): 165.000000 -> 239.000000 [183.875000]
Pingpong (1000000 doubles, 8000000 bytes): 16668.000000 -> 21351.000000 [16204.125000]
Alltoall (8 doubles/64 bytes), 1469.000000 -> 1677.000000
Alltoall (8 doubles/64 bytes), 7.000000 -> 17.000000 [Split Halves]
Alltoall (8192 doubles/65536 bytes), 520.000000 -> 553.000000
Alltoall (8192 doubles/65536 bytes), 264.000000 -> 279.000000 [Split Halves]
Alltoall (100000 doubles/800000 bytes), 3663.000000 -> 4222.000000
Alltoall (100000 doubles/800000 bytes), 2643.000000 -> 2972.000000 [Split Halves]
Barrier in 2.000000 -> 296.000000
Barrier in 4.000000 -> 56.000000 [Halfcomm]
Reduce    (min) (0 doubles, 0 bytes) 2.000000 -> 3.000000
AllReduce (min) (0 doubles, 0 bytes) 2.000000 -> 3.000000
AllReduce (min) (0 doubles, 0 bytes) 2.000000 -> 3.000000 [HalfComm]
AllReduce (add) (0 doubles, 0 bytes) 2.000000 -> 3.000000 [HalfComm]
Reduce    (min) (8 doubles, 64 bytes) 2.000000 -> 5.000000
AllReduce (min) (8 doubles, 64 bytes) 6.000000 -> 8.000000
AllReduce (min) (8 doubles, 64 bytes) 5.000000 -> 6.000000 [HalfComm]
AllReduce (add) (8 doubles, 64 bytes) 4.000000 -> 8.000000 [HalfComm]
Reduce    (min) (8192 doubles, 65536 bytes) 425.000000 -> 440.000000
AllReduce (min) (8192 doubles, 65536 bytes) 86.000000 -> 114.000000
AllReduce (min) (8192 doubles, 65536 bytes) 70.000000 -> 84.000000 [HalfComm]
AllReduce (add) (8192 doubles, 65536 bytes) 69.000000 -> 79.000000 [HalfComm]
Reduce    (min) (1000000 doubles, 8000000 bytes) 7940.000000 -> 8249.000000
AllReduce (min) (1000000 doubles, 8000000 bytes) 9931.000000 -> 10404.000000
AllReduce (min) (1000000 doubles, 8000000 bytes) 8739.000000 -> 9397.000000 [HalfComm]
AllReduce (add) (1000000 doubles, 8000000 bytes) 8889.000000 -> 9150.000000 [HalfComm]

Cartesian creation [8] 1234.000000 -> 1238.000000 [2 x 4]
Cartesian datatype 2.000000 -> 3.000000
Cartesian sendrecv iteration 0: 9.000000 -> 13.000000
Cartesian sendrecv iteration 1: 8.000000 -> 19.000000
Cartesian sendrecv iteration 2: 6.000000 -> 13.000000
Cartesian sendrecv iteration 3: 6.000000 -> 9.000000
Cartesian isend/irecv iteration 0: 6.000000 -> 7.000000
Cartesian isend/irecv iteration 1: 3.000000 -> 11.000000
Cartesian isend/irecv iteration 2: 4.000000 -> 9.000000
Cartesian isend/irecv iteration 3: 4.000000 -> 9.000000

Cartesian creation [10000] 51.000000 -> 55.000000 [2500 x 5000]
Cartesian datatype 2.000000 -> 3.000000
Cartesian sendrecv iteration 0: 36596.000000 -> 57419.000000
Cartesian sendrecv iteration 1: 907.000000 -> 19941.000000
Cartesian sendrecv iteration 2: 715.000000 -> 19406.000000
Cartesian sendrecv iteration 3: 596.000000 -> 825.000000
Cartesian isend/irecv iteration 0: 511.000000 -> 757.000000
Cartesian isend/irecv iteration 1: 416.000000 -> 607.000000
Cartesian isend/irecv iteration 2: 346.000000 -> 517.000000
Cartesian isend/irecv iteration 3: 289.000000 -> 587.000000

Cartesian creation [20000] 100.000000 -> 103.000000 [5000 x 10000]
Cartesian datatype 2.000000 -> 3.000000
Cartesian sendrecv iteration 0: 108685.000000 -> 113114.000000
Cartesian sendrecv iteration 1: 4045.000000 -> 6949.000000
Cartesian sendrecv iteration 2: 3095.000000 -> 3651.000000
Cartesian sendrecv iteration 3: 2903.000000 -> 3382.000000
Cartesian isend/irecv iteration 0: 2883.000000 -> 4329.000000
Cartesian isend/irecv iteration 1: 2584.000000 -> 4076.000000
Cartesian isend/irecv iteration 2: 2052.000000 -> 4207.000000
Cartesian isend/irecv iteration 3: 1760.000000 -> 4099.000000

Cartesian creation [40000] 79.000000 -> 83081.000000 [10000 x 20000]
Cartesian datatype 2.000000 -> 3.000000
Cartesian sendrecv iteration 0: 149916.000000 -> 399917.000000
Cartesian sendrecv iteration 1: 7132.000000 -> 257592.000000
Cartesian sendrecv iteration 2: 14364.000000 -> 192824.000000
Cartesian sendrecv iteration 3: 7386.000000 -> 185273.000000
Cartesian isend/irecv iteration 0: 13065.000000 -> 111588.000000
Cartesian isend/irecv iteration 1: 8051.000000 -> 177480.000000
Cartesian isend/irecv iteration 2: 5427.000000 -> 166412.000000
Cartesian isend/irecv iteration 3: 5807.000000 -> 165422.000000
```

A.2.3 Open MPI

```
Running on 8 nodes
Pingpong (0 doubles, 0 bytes): 21.219254 -> 208.854675 [53.226948]
Pingpong (0 doubles, 0 bytes): 1.907349 -> 77.009201 [11.861324]
Pingpong (100 doubles, 800 bytes): 10.967255 -> 15.974045 [11.742115]
Pingpong (1024 doubles, 8192 bytes): 38.146973 -> 93.936920 [48.011541]
Pingpong (8192 doubles, 65536 bytes): 147.104263 -> 221.014023 [164.270401]
Pingpong (1000000 doubles, 8000000 bytes): 14268.159866 -> 19603.967667 [15465.885401]
Alltoall (8 doubles/64 bytes), 6383.895874 -> 7472.991943
Alltoall (8 doubles/64 bytes), 7.152557 -> 11.205673 [Split Halves]
Alltoall (8192 doubles/65536 bytes), 364.065170 -> 389.099121
Alltoall (8192 doubles/65536 bytes), 160.932541 -> 217.199326 [Split Halves]
Alltoall (100000 doubles/800000 bytes), 3596.067429 -> 3659.963608
Alltoall (100000 doubles/800000 bytes), 2568.960190 -> 2754.211426 [Split Halves]
Barrier in 13.113022 -> 16896.009445
Barrier in 3.099442 -> 10.013580 [Halfcomm]
Reduce    (min) (0 doubles, 0 bytes) 6.914139 -> 16.212463
AllReduce (min) (0 doubles, 0 bytes) 0.000000 -> 2.145767
AllReduce (min) (0 doubles, 0 bytes) 0.953674 -> 2.145767 [HalfComm]
AllReduce (add) (0 doubles, 0 bytes) 0.953674 -> 2.145767 [HalfComm]
Reduce    (min) (8 doubles, 64 bytes) 4275.798798 -> 4333.972931
AllReduce (min) (8 doubles, 64 bytes) 5.960464 -> 9.059906
AllReduce (min) (8 doubles, 64 bytes) 3.814697 -> 7.152557 [HalfComm]
AllReduce (add) (8 doubles, 64 bytes) 3.099442 -> 7.152557 [HalfComm]
Reduce    (min) (8192 doubles, 65536 bytes) 31.948090 -> 99.897385
```

```
AllReduce (min) (8192 doubles, 65536 bytes) 134.944916 -> 144.958496
AllReduce (min) (8192 doubles, 65536 bytes) 72.002411 -> 82.015991 [HalfComm]
AllReduce (add) (8192 doubles, 65536 bytes) 82.969666 -> 87.022781 [HalfComm]
Reduce    (min) (1000000 doubles, 8000000 bytes) 4830.121994 -> 5487.918854
AllReduce (min) (1000000 doubles, 8000000 bytes) 9468.078613 -> 9511.947632
AllReduce (min) (1000000 doubles, 8000000 bytes) 7586.956024 -> 7669.210434 [HalfComm]
AllReduce (add) (1000000 doubles, 8000000 bytes) 7786.035538 -> 7843.017578 [HalfComm]

Cartesian creation [8] 69802.999496 -> 69808.959961 [2 x 4]
Cartesian datatype 1608.133316 -> 2157.926559
Cartesian sendrecv iteration 0: 372.886658 -> 384.092331
Cartesian sendrecv iteration 1: 5.960464 -> 19.073486
Cartesian sendrecv iteration 2: 4.053116 -> 11.920929
Cartesian sendrecv iteration 3: 6.198883 -> 11.920929
Cartesian isend/irecv iteration 0: 293.970108 -> 368.118286
Cartesian isend/irecv iteration 1: 5.960464 -> 70.810318
Cartesian isend/irecv iteration 2: 5.960464 -> 14.066696
Cartesian isend/irecv iteration 3: 5.006790 -> 10.967255

Cartesian creation [10000] 40.054321 -> 42.915344 [2500 x 5000]
Cartesian datatype 1.907349 -> 4.053116
Cartesian sendrecv iteration 0: 39411.067963 -> 56400.060654
Cartesian sendrecv iteration 1: 1141.071320 -> 18459.081650
Cartesian sendrecv iteration 2: 575.065613 -> 823.974609
Cartesian sendrecv iteration 3: 639.915466 -> 681.877136
Cartesian isend/irecv iteration 0: 429.153442 -> 555.992126
Cartesian isend/irecv iteration 1: 416.994095 -> 546.932220
Cartesian isend/irecv iteration 2: 349.998474 -> 550.985336
Cartesian isend/irecv iteration 3: 411.987305 -> 634.908676

Cartesian creation [20000] 72.956085 -> 91.075897 [5000 x 10000]
Cartesian datatype 9.059906 -> 13.113022
Cartesian sendrecv iteration 0: 81027.030945 -> 115333.080292
Cartesian sendrecv iteration 1: 4553.079605 -> 38539.171219
Cartesian sendrecv iteration 2: 2707.958221 -> 2817.869186
Cartesian sendrecv iteration 3: 2341.985703 -> 2452.135086
Cartesian isend/irecv iteration 0: 2022.027969 -> 2608.060837
Cartesian isend/irecv iteration 1: 1780.986786 -> 2400.159836
Cartesian isend/irecv iteration 2: 1901.865005 -> 2679.824829
Cartesian isend/irecv iteration 3: 1763.820648 -> 2293.109894

Cartesian creation [40000] 72.002411 -> 97.036362 [10000 x 20000]
Cartesian datatype 10.013580 -> 13.113022
Cartesian sendrecv iteration 0: 156381.130219 -> 232652.902603
Cartesian sendrecv iteration 1: 13305.902481 -> 89359.045029
Cartesian sendrecv iteration 2: 14761.924744 -> 15942.096710
Cartesian sendrecv iteration 3: 14080.047607 -> 15033.960342
Cartesian isend/irecv iteration 0: 10293.960571 -> 14144.897461
Cartesian isend/irecv iteration 1: 11479.139328 -> 15923.023224
Cartesian isend/irecv iteration 2: 10748.863220 -> 14539.957047
Cartesian isend/irecv iteration 3: 10757.923126 -> 14139.175415
```

Appendix B

All-to-all

B.1 Code

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#define NRM 500
#define NRN 500
#define REPEATS 1000
double **createDoubleMatrix(int n1, int n2);

int main ( int argc , char *argv[]) {
        int mpi_err;
        int np, rank;
        int a_rows, a_cols;
        int i, j;
        double **m;
        double *rray, *sray;
        int ssize, rsize;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Status status;

        double *t;
        if(rank==0) t = (double *)malloc(sizeof(double)*REPEATS*2);
        a_rows = NRM / np;
        a_cols = NRN / np;
        m = createDoubleMatrix(NRM, a_cols);
        srand(rank);

        int elements = 0;

        for(i = 0; i < NRM; i++) {
                for(j = 0; j < a_cols; j++) {
                        m[i][j] = i;
                        elements++;
                }
        }

        ssize = rsize = 0;
        for(i = 0; i < np; i++) {
                ssize = ssize + a_cols*a_rows;
                rsize = rsize + a_cols*a_rows;
        }
        sray = (double*)malloc(sizeof(double)*ssize);
        rray = (double*)malloc(sizeof(double)*rsize);

        if(sray == NULL || rray == NULL) perror("send and receive arrays");

        int k = 0;
        int l = 0;
        for(k = 0; k < np; k++) {
                int offset = (k * a_rows);
                for(i = 0; i < a_rows; i++) {
                        for(j = 0; j < a_cols; j++) {
                                sray[l] = m[i+offset][j];
                                l++;
                        }
                }
        }
        k = 0;

        for(i = 0; i < REPEATS; i++) {
                if(rank==0) t[k++] = MPI_Wtime();
                MPI_Alltoall(sray, a_cols*a_rows, MPI_DOUBLE,
                        rray, a_cols*a_rows, MPI_DOUBLE, MPI_COMM_WORLD);
                if(rank==0) t[k++] = MPI_Wtime();
                MPI_Barrier(MPI_COMM_WORLD);
        }

        if(rank == 0) {
                double total = 0.0;
                double tmp = 0.0;
                double min = 1000.0;
                double max = 0.0;
                int minrun, maxrun;
                k = 0;

                for(i = 0; i < REPEATS; i++) {
                        tmp = t[k+1] - t[k];
                        k += 2;
                        if(tmp >= max) {
                                max = tmp;
                                maxrun = i;
                        }else if(tmp <= min) {
                                min = tmp;
                                minrun = i;
                        }

                        total += tmp;
                }

                printf("total [%i runs]: %e avg: %e (max: %e[%i]/min: %e[%i])\n", i, total, (double)total/i, max, m
        }

        l = 0;
        for(k = 0; k < np; k++) {
                int offset = (k * a_rows);
                for(j = 0; j < a_cols; j++) {
                        for(i = 0; i < a_rows; i++) {
                                m[i+offset][j] = rray[l];
                                l++;
                        }
                }
        }

        MPI_Finalize();

        return 0;
}
```

```
double **createDoubleMatrix(int n1, int n2) {
        double **a;
        int i, j;
        a = (double **)malloc(n1 * sizeof(double *));
        a[0] = (double *)malloc(n1*n2*sizeof(double));
        for(i = 1; i < n1; i++ )  {
                a[i] = a[i-1] + n2;
        }

        for(i = 0; i < n1; i++) {
                for(j = 0; j < n2; j++) {
                        a[i][j] = 0.0f;
                }
        }

        return a;
}
```

B.2 Results

Table B.1: All-to-all, small sized messages(400x50 doubles)

Impl	Iter	Tot	Avg	Max	Min
MPICH	1000	8.176477E-01	8.176477E-04	1.471901E-02	6.039143E-04
MPICH	1000	8.063095E-01	8.063095E-04	1.307487E-02	3.631115E-04
MPICH	1000	8.171675E-01	8.171675E-04	1.401806E-02	6.020069E-04
MPICH	1000	7.991631E-01	7.991631E-04	1.793861E-03	3.600121E-04
MPICH	5000	4.013466E+00	8.026933E-04	4.582882E-03	4.930496E-04
MPICH	5000	4.291058E+00	8.582116E-04	1.628184E-02	5.290508E-04
MPICH	5000	4.222188E+00	8.444376E-04	1.553488E-02	2.648830E-04
MPICH	5000	4.101371E+00	8.202743E-04	2.758789E-02	5.049706E-04
MPICH	10000	8.080390E+00	8.080390E-04	1.489401E-02	3.039837E-04
MPICH	10000	7.984199E+00	7.984199E-04	1.318216E-02	4.351139E-04
MPICH	10000	8.611192E+00	8.611192E-04	4.044199E-02	4.229546E-04
MPICH	10000	8.231399E+00	8.231399E-04	1.286483E-02	2.000332E-04
MVAPICH	1000	5.766840E-01	5.766840E-04	1.302000E-03	3.720000E-04
MVAPICH	1000	5.752150E-01	5.752150E-04	1.151000E-03	3.800000E-04
MVAPICH	1000	5.648260E-01	5.648260E-04	1.131000E-03	3.850000E-04
MVAPICH	1000	5.779820E-01	5.779820E-04	1.089000E-03	3.740000E-04
MVAPICH	5000	2.846303E+00	5.692606E-04	1.235900E-02	3.580000E-04
MVAPICH	5000	2.872527E+00	5.745054E-04	1.385400E-02	3.600000E-04
MVAPICH	5000	2.856492E+00	5.712984E-04	1.378400E-02	3.650000E-04
MVAPICH	5000	2.853415E+00	5.706830E-04	1.160700E-02	3.610000E-04
MVAPICH	10000	5.784503E+00	5.784503E-04	2.397600E-02	3.630000E-04
MVAPICH	10000	5.732647E+00	5.732647E-04	1.374400E-02	3.610000E-04
MVAPICH	10000	5.722037E+00	5.722037E-04	1.383900E-02	3.670000E-04
MVAPICH	10000	5.622556E+00	5.622556E-04	2.344000E-02	3.620000E-04
Open MPI	1000	5.006273E-01	5.006273E-04	8.707047E-03	4.441738E-04
Open MPI	1000	4.915588E-01	4.915588E-04	1.427197E-02	4.260540E-04
Open MPI	1000	5.260437E-01	5.260437E-04	3.931093E-02	4.370213E-04
Open MPI	1000	4.763649E-01	4.763649E-04	3.191948E-03	4.179478E-04
Open MPI	5000	2.457770E+00	4.915540E-04	4.745388E-02	4.379749E-04
Open MPI	5000	2.382580E+00	4.765161E-04	1.162696E-02	4.270077E-04
Open MPI	5000	2.654265E+00	5.308530E-04	2.072661E-02	4.370213E-04
Open MPI	5000	2.374375E+00	4.748749E-04	1.832199E-02	4.239082E-04
Open MPI	10000	4.768914E+00	4.768914E-04	6.028891E-03	4.169941E-04
Open MPI	10000	4.798383E+00	4.798383E-04	2.115512E-02	4.220009E-04
Open MPI	10000	4.762806E+00	4.762806E-04	4.543686E-02	4.241467E-04
Open MPI	10000	4.699058E+00	4.699058E-04	2.197981E-03	4.210472E-04

Table B.2: All-to-all, medium sized messages(1000x1000 doubles)

Impl	Iter	Tot	Avg	Max	Min
MPICH	1000	8.543125e+02	8.543125e-01	2.437500e+00	1.445312e-01
MPICH	1000	9.640234e+02	9.640234e-01	2.355469e+00	3.476562e-01
MPICH	1000	9.796133e+02	9.796133e-01	2.925781e+00	2.968750e-01
MPICH	1000	9.704492e+02	9.704492e-01	2.550781e+00	1.484375e-01
MPICH	5000	5.161012e+03	1.032202e+00	2.808594e+00	1.679688e-01
MPICH	5000	5.153426e+03	1.030685e+00	2.847656e+00	1.601562e-01
MPICH	5000	5.131906e+03	1.026381e+00	3.035156e+00	2.929688e-01
MPICH	5000	5.149297e+03	1.029859e+00	3.089844e+00	1.796875e-01
MPICH	10000	8.793560e+03	8.793560e-01	1.679313e+01	4.113698e-02
MPICH	10000	7.292942e+03	7.292942e-01	1.887195e+01	3.765798e-02
MPICH	10000	8.637271e+03	8.637271e-01	1.276544e+01	4.608297e-02
MPICH	10000	8.008461e+03	8.008461e-01	1.327548e+01	4.590607e-02
MVAPICH	1000	4.820071e+01	4.820071e-02	7.076800e-02	3.094600e-02
MVAPICH	1000	4.894080e+01	4.894080e-02	6.652600e-02	3.794900e-02
MVAPICH	1000	4.880722e+01	4.880722e-02	7.225700e-02	3.604300e-02
MVAPICH	1000	4.907290e+01	4.907290e-02	6.702600e-02	3.079800e-02
MVAPICH	5000	2.461737e+02	4.923474e-02	7.342500e-02	3.061900e-02
MVAPICH	5000	2.439453e+02	4.878905e-02	7.436500e-02	3.049300e-02
MVAPICH	5000	2.442553e+02	4.885106e-02	7.311600e-02	3.048000e-02
MVAPICH	5000	2.458720e+02	4.917441e-02	7.400000e-02	3.108600e-02
MVAPICH	10000	4.914969e+02	4.914969e-02	7.468300e-02	3.080700e-02
MVAPICH	10000	4.911269e+02	4.911269e-02	7.392000e-02	3.057100e-02
MVAPICH	10000	4.904774e+02	4.904774e-02	7.433300e-02	3.054600e-02
MVAPICH	10000	4.898499e+02	4.898499e-02	7.359700e-02	3.058200e-02
Open MPI	1000	4.704773e+01	4.704773e-02	6.683207e-02	4.432511e-02
Open MPI	1000	4.557816e+01	4.557816e-02	5.524898e-02	4.368615e-02
Open MPI	1000	4.713416e+01	4.713416e-02	1.480470e-01	3.830099e-02
Open MPI	1000	4.714683e+01	4.714683e-02	1.295910e-01	4.472303e-02
Open MPI	5000	2.317214e+02	4.634429e-02	7.062197e-02	3.633189e-02
Open MPI	5000	2.144124e+02	4.288247e-02	6.678009e-02	3.964806e-02
Open MPI	5000	2.126782e+02	4.253565e-02	6.506300e-02	4.104400e-02
Open MPI	5000	2.119173e+02	4.238347e-02	6.303906e-02	4.039907e-02
Open MPI	10000	4.616412e+02	4.616412e-02	6.941700e-02	3.950596e-02
Open MPI	10000	4.678542e+02	4.678542e-02	2.297812e-01	3.713012e-02
Open MPI	10000	4.700857e+02	4.700857e-02	6.936812e-02	3.987813e-02
Open MPI	10000	4.635596e+02	4.635596e-02	7.590199e-02	4.142213e-02

Table B.3: All-to-all, large sized messages(8000x8000 doubles)

Impl	Iter	Tot	Avg	Max	Min
MVAPICH	1000	3.181639e+03	3.181638e+00	4.227398e+00	3.070103e+00
MVAPICH	1000	3.181654e+03	3.181654e+00	6.010840e+00	3.105738e+00
MVAPICH	1000	3.189418e+03	3.189418e+00	6.887669e+00	3.082856e+00
MVAPICH	1000	3.187557e+03	3.187557e+00	5.065821e+00	3.103707e+00
MVAPICH	5000	1.600605e+04	3.201210e+00	5.382317e+00	3.050804e+00
MVAPICH	5000	1.590302e+04	3.180604e+00	6.897454e+00	2.940583e+00
MVAPICH	5000	1.579295e+04	3.158591e+00	5.359114e+00	2.946226e+00
MVAPICH	5000	1.590908e+04	3.181815e+00	6.863655e+00	3.036344e+00
Open MPI	1000	3.039166e+03	3.039166e+00	3.960890e+00	2.772568e+00
Open MPI	1000	2.962924e+03	2.962924e+00	5.607667e+00	2.874455e+00
Open MPI	1000	3.017786e+03	3.017786e+00	3.146511e+00	2.965070e+00
Open MPI	1000	2.962210e+03	2.962210e+00	3.327820e+00	2.812816e+00
Open MPI	5000	1.352857e+04	2.705714e+00	3.590074e+00	2.606230e+00
Open MPI	5000	1.342850e+04	2.685701e+00	4.272410e+00	2.603990e+00
Open MPI	5000	1.502338e+04	3.004676e+00	8.440918e+00	2.944116e+00
Open MPI	5000	1.408224e+04	2.816447e+00	3.058857e+00	2.582932e+00

Appendix C

Reduce

C.1 Code

```
#include <stdlib.h>
#include <stdio.h>
#include "/home/abach/openmpi-impl/include/mpi.h"
#define NRM 1000
#define NRN 8000
#define REPEATS 1000
double **createDoubleMatrix(int n1, int n2);

int main ( int argc , char *argv[]) {
        int mpi_err;
        int np, rank;
        int a_rows, a_cols;
        int i, j;
        double **m;
        double *rray, *sray, *fillray;
        int ssize, rsize;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Status status;
        double *t;

        if(rank==0) {
                t = (double *)malloc(sizeof(double)*REPEATS*2);
                if(t == NULL) perror("t");
        }

        a_rows = NRM / np;
        a_cols = NRN / np;

        m = createDoubleMatrix(NRM, a_cols);

        srand(rank);

        int elements = 0;
        for(i = 0; i < NRM; i++) {
                for(j = 0; j < a_cols; j++) {
                        m[i][j] = i;
                        elements++;
                }
        }

        ssize = rsize = 0;
```

```
for(i = 0; i < np; i++) {
        ssize = ssize + a_cols*a_rows;
        rsize = rsize + a_cols*a_rows;
}

fillray = (double*)malloc(sizeof(double)*ssize);
if(fillray == NULL) perror("fillray");
rray = (double*)malloc(sizeof(double)*rsize);

int k = 0;
int l = 0;
int ii = 0;
int jj = 0;
double result = 0.0f;
for(ii = 0; ii < REPEATS; ii++) {
        free(rray);
        sray = (double*)malloc(sizeof(double)*ssize);
        rray = (double*)malloc(sizeof(double)*rsize);
        if(sray == NULL || rray == NULL) perror("send and receive arrays");
        for(jj = 0; jj < np; jj++) {
                int offset = (jj * a_rows);
                for(i = 0; i < a_rows; i++) {
                        for(j = 0; j < a_cols; j++) {
                                sray[l] = m[i+offset][j] * rand();
                                l++;
                        }
                }
        }

        l = 0;
        for(jj = 0; jj < np; jj++) {
                int offset = (jj * a_rows);
                for(i = 0; i < a_rows; i++) {
                        for(j = 0; j < a_cols; j++) {
                                fillray[l] = m[i+offset][j];
                                l++;
                        }
                }
        }

        if(rank==0) t[k++] = MPI_Wtime();
        MPI_Reduce(sray, &result, a_cols*a_rows, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);
        if(rank==0) t[k++] = MPI_Wtime();
        MPI_Barrier(MPI_COMM_WORLD);

        free(sray);
        l = 0;
}

if(rank == 0) {
        double total = 0.0;
        double tmp = 0.0;
        double min = 1000.0;
        double max = 0.0;
        double remove = 0.0;
        int minrun, maxrun;
        k = 0;
        for(i = 0; i < REPEATS; i++) {
                tmp = t[k+1] - t[k];
                k += 2;
                if(tmp >= max) {
                        max = tmp;
                        maxrun = i;
                }else if(tmp <= min) {
                        min = tmp;
                        minrun = i;
                }
```

```
                        total += tmp;
                        if(i <= 5 || i >= REPEATS-5) {
                                remove += tmp;
                        }
                }

                printf("total [%i runs]: %e avg: %e (max: %e[%i]/min: %e[%i])\n", i, total, (double)total/i, max, maxrun, min, min
                printf("removed first and last 5: total: %e avg: %e\n", total-remove, (double)(total-remove)/(i-5));
        }

        l = 0;
        for(k = 0; k < np; k++) {
                int offset = (k * a_rows);
                for(j = 0; j < a_cols; j++) {
                        for(i = 0; i < a_rows; i++) {
                                m[i+offset][j] = rray[l];
                                l++;
                        }
                }
        }
        MPI_Finalize();

        return 0;

}


double **createDoubleMatrix(int n1, int n2) {
        double **a;
        int i, j;
        a = (double **)malloc(n1 * sizeof(double *));
        a[0] = (double *)malloc(n1*n2*sizeof(double));
        for(i = 1; i < n1; i++ )  {
                a[i] = a[i-1] + n2;
        }

        for(i = 0; i < n1; i++) {
                for(j = 0; j < n2; j++) {
                        a[i][j] = 0.0f;
                }
        }

        return a;
}
```