

Chapter 1

INTRODUCTION

Clusters of workstations are commonly used among engineers and domain scientists because clusters have enormous processing power and relatively low cost. The major drawback of cluster-based parallel computing as compared to using a shared memory multiprocessor machine is the network delay induced by the node interconnecting technology of clusters. Several interconnection technologies have been developed with the goal of improving cluster parallel performance by providing specialized low latency, high bandwidth, networks for clusters.

Unfortunately, although specialized networks perform better than their legacy counterparts, the low latency and high bandwidth that is associated with them is more often achieved in benchmarks than in real world parallel applications. Only a small percentage of existing scientific codes can scale to hundreds of CPUs without suffering significant loss of efficiency due to communication overheads. In regular codes, such as FFT, the main impediment to scalability is the communication overhead as the number of nodes increases.

In this thesis, I experimentally evaluate a transformation to overlap communication-computation in parallel programs in order to hide communication overhead. The transformation was developed by Danalis et. al. [5] at the University of Delaware. The major goal of this thesis is to evaluate the transformation technique to demonstrate how

the performance of real-world parallel applications can be improved by applying the transformation.

Section 2 provides background information and state of the art, such as parallel programming models and cluster networks as well as communication libraries. Section 3 describes the optimization technique and several transformation strategies used in this work. Sections 4 and 5 provide experimental results, analysis, and conclusions.

Chapter 2

BACKGROUND AND STATE OF THE ART

2.1 Programming Style

There exist several parallel programming models, including shared memory, message passing, data parallel, and hybrids. The parallel programming models exist atop abstractions of hardware and memory architectures. In the shared memory model, tasks share an address space, which they can read and write asynchronously. In the message passing model, each task has its own memory space, and the tasks exchange messages through send and receive operations. In the data parallel programming model, a set of tasks collectively process partitions of the same data structure. Hybrid models enable message passing between processes on different nodes and threaded computation with shared memory communication within a multiprocessor node.

MPI is a message passing communication specification. It is the most common method for writing parallel programs [1]. There are many MPI implementations, such as MPICH [2], MPICH-GM [3], LAM/MPI [4], and IBM's MPI [5]. This thesis focuses on optimizations in MPI-based programs, which are the most common programs in loosely coupled parallel computers, such as clusters.

2.2 Cluster Networks and Communication Libraries

A cluster is a parallel computer which consists of many machines interconnected

by a high-speed network. As PCs and lightweight workstation performance and accommodated network speeds grow, cluster computing becomes more cost effective and scalable. Several interconnection technologies have been developed with the goal of improving cluster message-passing performance by providing specialized, low latency, high bandwidth networks for clusters.

MPICH is a freely available, portable implementation of the MPI standard. It contains a complete implementation of version 1.2 of the MPI Standard and also significant parts of MPI-2. MPICH-GM is MPICH ported by Myricom to use their GM library for their Myrinet networking architecture. GM is a user-level communication library and protocol that runs over the Myrinet network and provides a reliable ordered delivery of packets with low latency and high bandwidth. Our research group has developed a Custom Library built on top of the GM API to ease usage of the GM API and to provide a Fortran interface (the GM API does not support Fortran) while keeping the overhead minimum. This custom library enables us to issue one-sided transfer operations. Such operations utilize the Remote Direct Memory Access (RDMA) functionality of the network interface and exhibit low latency, high throughput and minimum of the host CPU.

In order to avoid extra control messages in communication, I adopted memory polling synchronization. Each send transfers the requested data plus two more numbers at the two edges of the data buffer that are used as flags. The value of these flags is polled by the receiver when transfer completion needs to be confirmed so that additional control messages are avoided. This busy-wait synchronization might be a waste of CPU time, and periodically polling memory induces memory contention between the host CPU and network processor. By setting a proper polling period, however, I could moderate these over-

heads.

Chapter 3

OPTIMIZATION TECHNIQUE

3.1 Overview

In this section, I describe the optimizing transformation, developed by Danalis et. al [5], to restructure the computation loop of parallel MPI codes into smaller tiles that perform part of the computation and the corresponding communication. Non-blocking I/O is used to pipeline the execution of consecutive tiles in order for the communication of a tile to be overlapped with the computation of the next. This transformation can be applied under certain conditions. The original code should contain a loop nest, where each iteration of one of the loops has independent data access which means that the outcome of each iteration does not depend on the previous iterations. Although a data dependent loop can be restructured into a form that can be transformed similarly to loops with no inter iteration dependencies, I make the above assumption for simplicity reasons.

The two most important parameters for the performance of the transformed code are the tile size (K) and the depth of the tile pipeline (D); in other words, the number of future tiles that the communication of each tile is overlapped with. K and D are the two parameters we define to find the best form of the transformed code. The transformed program will initiate a send in every K iterations and blocks for the previous D 'th send. K is relevant to the number of sends and the data size of each send. D is relevant to the size of resources to send and receive data, and also D specifies the delay (in tiles) after which the

application will block waiting for a network operation to complete.

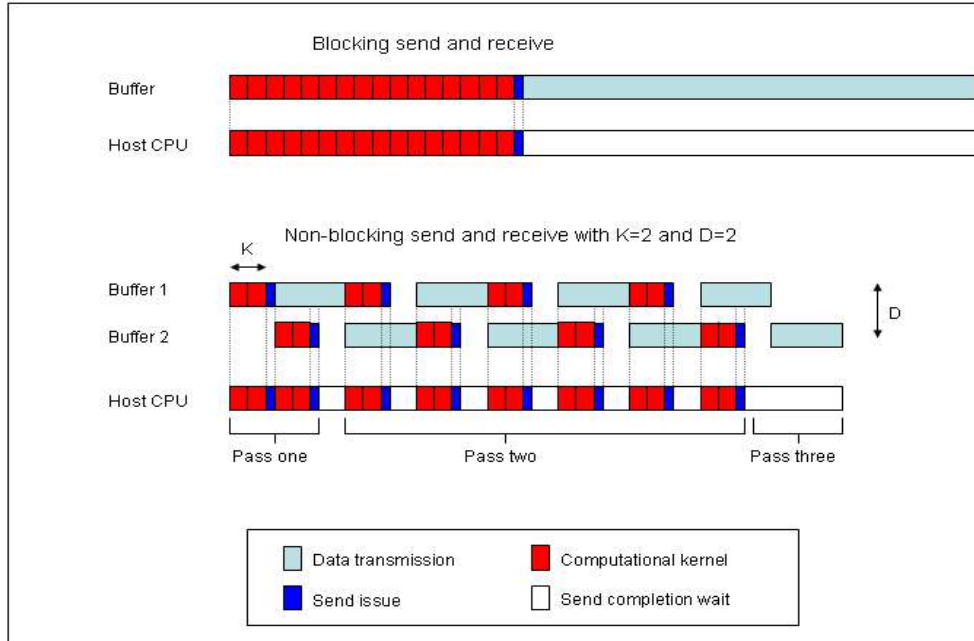


Figure 3.1 Overlapping Communication and computation model with $D=2$

Figure 3.1 depicts the communication under the two scenarios of computation-communication overlap. I assume there are 16 iterations in the loop of the computational kernel. In the case of blocking send and receive (top diagram in Figure 3.1), the entire computational kernel is performed at once, and then the whole data are transmitted at the end. There is one send issue overhead, and the host CPU has to wait for the completion of the whole data transmission. In the case of the non-blocking send and receive with $K=2$ and $D=2$ (bottom diagram in Figure 3.1), each send contains two computational kernel iterations, and two sending buffers are used. Since the computational kernel is divided into eight tiles (indicated by the eight data transmission blocks), the program requires eight send issue overheads. This is a drawback of having a small K value. However, part of the

data transmission (communication) is overlapped with kernel computation.

One can easily expand this idea by setting various values of K and D . The transformed code has three passes. In Pass One, D number of kernel computations and send issues are performed without waiting for send completion. In Pass Two, kernel computation, send issue, and previous send completion wait are performed in round robin manner. In Pass Three, the program waits for all the send completions issued in Pass Two.

By setting the optimal tile size (K) and pipeline depth (D), the program can minimize the synchronization overhead. However, determining the best value for these parameters is not a trivial task as there are several tradeoffs and details associated with them. In particular, increasing K leads to a larger tile size which corresponds to fewer and larger messages. This is a desirable effect since the fixed overhead of a single transfer operation is amortized over larger transfers. At the same time, higher average bandwidth is achieved, as larger messages experience higher transfer rates. On the other hand, as K increases, the size of the last tile increases and that communication cannot be overlapped with computation. D controls the pipelining of the tiles. Higher values of D increase the width of pipelining. Higher D values also require more network resources because there are more concurrent outstanding messages. The sizes of sending and receiving buffers are proportional to the D value, and the operating system may not have sufficient buffers that can be transferred using DMA to manage high D values. One may argue that MPICH-GM does not limit sending or receiving buffer size, so the D value can be as high as desired. In fact, MPICH-GM internally copies data from the non-DMA enabled buffer to the DMA enabled buffer to send or receive data.

If the network interface does not have a co-processor dedicated to data transmis-

sion, there is no benefit of applying the discussed transformation because the host CPU is busy with work for data transmission. I have experimentally shown this fact with a cluster interconnected by Gigabit Ethernet. Our experimental results support the observation that non-blocking and asynchronous I/O are not the same. Non-blocking I/O can be supported by a library regardless of the network hardware. In a programmed communication environment though, even if the call to the network operation returns (instead of blocking), the computation cannot proceed because the CPU and the memory are busy performing the transfer. Truly asynchronous I/O is achieved only if the network hardware can perform the transfer on its own, letting the host processor continue with its computation. Actually in the case where the main processor is performing the transfer, cache pollution can even turn communication-computation overlapping into a non-profitable transformation.

3.2 Transformation Strategies

In this thesis, I experimentally evaluate five different strategies for communication, based on the computation-communication overlap transformation. Presumably the original codes use `MPI_ALLTOALL` because it is the simplest way to exchange data in applications like FFT or sort programs.

3.2.1 Blocking Communications

- **MPICH `MPI_ALLTOALL`** - The '`MPI_ALLTOALL()`' collective communication with blocking I/O function is used to perform the communication after the entire computation. The implementation of MPI is MPICH, which is widely used, but not specifically designed for an RDMA-enabled network.
- **MPICH-GM `MPI_ALLTOALL`** - The '`MPI_ALLTOALL()`' collective commu-

nication with blocking I/O function is used in the same way as the first scheme, but the MPI implementation is MPICH-GM, which is provided by the Myricom and uses the advanced features of the network technology (e.g., RDMA).

3.2.2 Non-blocking Communications

- **MPICH MPI_ISEND** – The computation is reorganized into independent tiles and communication is performed in each tile. The 'MPI_ISEND()' and 'MPI_IRECV()' are used to issue send and receive of the tiles. The 'MPI_WAIT-ALL()' is used for send and receive completions.
- **MPICH-GM MPI_ISEND** – This model also uses the 'MPI_ISEND()', but the MPI implementation is MPICH-GM.

Custom Library, one-sided I/O - the communication is handled by a low level library built directly on top of GM and providing one-sided I/O.

3.3 The Transformation Process

Figures 3.2 and 3.3 display original and transformed pseudo-code. Be aware that the function calls, such as `computational_kernel(m)`, `copy_buffer_to_send_to_process(j)`, and `mpi_alltoall()`, have abstract meaning, and they do not have to be actual functions or subroutines.

- `computational_kernel(m)` – This function computes the m^{th} part of the kernel.
- `copy_buffer_to_send_to_process(j)` – This function copies data to the send buffer to send to process j .
- `isend(j, request)` – This function initiates non-blocking send to process j , and the

context of this operation is stored in *request* variable. The *request* variable is used to keep track of the operation status. *isend* is similar to MPI's `MPI_ISEND()` function.

- `irecv(j, request)` – This function initiates receive of a message from process *j*. The *request* variable is used for the same purpose as *request* in `isend()`.
- `waitall(requests)` – This function will wait for the completion of all the *request* (s), similar to MPI's `MPI_WAITALL()` function.
- `alltoall()` - ALLTOALL data transformation in which each process sends distinct data to the other processes. This is similar to MPI's `MPI_ALLTOALL()` function.

```
! kernel computation
do x = 1, lz
  computational_kernel(x)
end do

! pack data to send
do j = 0, nproc-1
  if(j .ne. myrank) then
    copy_buffer_to_send_to_process(j)
  endif
end do

! data exchange by ALLTOALL
alltoall()
```

Figure 3.2 Original pseudo-code

The pseudo-code in Figure 3.2 is the abstract code we want to transform. It contains a computational kernel loop which has *lz* iterations, and each iteration is independent from the previous iterations. After the kernel computation, the program packs data to transmit to the other processes. MPI's ALLTOALL collective communication is per-

formed at the end. Note that communication and computation tasks are entirely separated, and blocking I/O and `MPI_ALLTOALL()` are used.

```

iy = lz/K ! assume lz is divisible by K
! pass one - compute kernel, and issue
! send and receive.
do i=1,D
  start_m=(i-1)*K
  do m=start_m+1,start_m+K
    computational_kernel(m)
  enddo
  do j=0,nproc-1
    if(j .ne. myrank) then
      copy_buffer_to_send_to_process(j)

      ! issue send and receive
      isend(j, request(1, j, i))
      irecv(j, request(2, j, i))
    endif
  enddo
enddo

! pass two - compute kernel, wait for send
! & receive completion, and issue send and
! receive.
do i=D+1,iy
  ri = mod(i-1,D)+1
  start_m=(i-1)*K
  do m=start_m+1,start_m+K
    computational_kernel(m)
  enddo

  call waitall(request(1:2, :, ri))

  do j=0,nproc-1
    if(j .ne. myrank) then
      copy_data_from_receive_buffer(j)
      copy_buffer_to_send_to_process(j)

      ! issue send and receive
      isend(j, request(1, j, ri))
      irecv(j, request(2, j, ri))
    endif
  enddo
enddo

! pass three - wait for send & receive
! completion.
do i=iy-D+1,iy
  ri = mod(i-1,D)+1
  waitall(request(1:2, :, ri))

  do j=0,nproc-1
    if(j .ne. myrank) then
      copy_data_from_receive_buffer(j)
    endif
  enddo
enddo

```

Figure 3.3 Transformed pseudo-code

Figure 3.3 shows the pseudo-code that is the result of the transformation. To reduce the complexity of the transformation, we assume the number of iterations, variable 'lz,' in the kernel loop is divisible by 'K.' One can easily fix this problem by adding additional housekeeping routines at the end of the code.

The transformed code has three passes. Pass One has a loop that contains D iterations. In each iteration, the program computes K kernel iterations, copies the result of computation to the send buffers, and issues non-blocking send and receive, to exchange the result of the computation with the other processes. The send requests are stored in the array 'request.' In Pass Two, at the first step, the program computes K iterations of the kernel, and the program waits for the completion of the send operation from the previous Dth iteration. As the last step, the program issues the send and receive operations for the resulting data of the computational kernel. Both steps in Pass Two are repeated in the loop. Send and receive buffers are also repeatedly used in loop. The loop in Pass Two ends when all the kernels are computed. In Pass Three, the program waits for the completion of the remaining send and receive operations which were issued in Pass Two, but not completed in Pass Two.

Chapter 4

EXPERIMENTAL STUDY

4.1 Research Questions

I conducted several experiments to evaluate the described communication strategies in two different application codes. The particular questions targeted by this study are as follows.

1. Is there a significant difference in performance between MPICH and MPICH-GM when using a Myrinet cluster?
2. What performance gain can be attained when non-blocking I/O is overlapped with the computation compared to the case where communication is not overlapped with the computation?
3. How does the specialization of the MPI library to a particular interconnecting network affect the answer to the previous question?
4. How sensitive is the overhead of each communication strategy to the message size?

4.2 Experimental Setup

4.2.1 Experimental Environment

The cluster that I used for the experiments consists of 20 Sun Microsystems Ultra Enterprise 450 machines, each with 4 250MHz Ultra-II processors. Eight nodes of the cluster have 1GB of memory 4-way interleaved. The other 12 nodes have 512MB of

memory 4-way interleaved. Each of the 20 nodes has a 4GB system disk. The interconnecting network is Myrinet. The Myrinet cards are model M2MPCI32b (LANAI 0x0403, 32-bit) and each card has 512K SRAM. Despite the absolute performance of this hardware, I assert that the relationships between processor and I/O speeds are such that our approach remains applicable for faster processors and interconnection networks.

I executed experiments varying several parameters, in order to compare their relative significance. In particular, for every implementation, I varied the number of processors (NP), the tile size (K), the depth of the tile execution pipeline (D), and the problem size. The dependent variable that I measured is the execution time. To compare the different communication strategies, I use the minimum execution time achieved by each strategy, as the random error in execution time can only be positive and therefore the minimum value should be the most accurate.

4.2.2 Experimental Codes

The first application, *magnet*, in the experiment is used for investigation of magneto-hydrodynamic turbulence through spectral methods. The second application that I used is Particle Mesh Ewald Molecular Dynamics (PMEMD) simulations, which is a tool used for theoretical study of biological molecules. PMEMD computes time dependent behavior of the molecular system. PMEMD also provides detailed information about the fluctuations and conformational changes of proteins and nucleic acids. This method is now routinely used to investigate the structure, dynamics and thermodynamics of biological molecules and their complexes. Both *magnet* and PMEMD are written in Fortran 90. Despite the differences in the structure, functionality, and programming styles of *magnet*

and PMEMD, the transformations were performed by the same process. I only manipulate the communication and the computation loop that “generates” the data to be sent. In both cases, the transformed code contained loops executing one or two dimensional FFT using the FFTW library.

In both cases, all the codes that I transformed were within one subroutine or function. Therefore, I timed only the parts that I transformed to measure the real effect of the transformations. Because I applied the same transformation process to both applications despite major differences between the two applications, I believe that the transformation technique is fairly general, and can be applied to many other applications.

The applications were compiled with mpich-1.2.5, gm-mpich-1.2.6.13, and gm-1.2.3 according to the needs of each communication scheme. The custom library, which was developed over GM, provides minimum abstraction in order for Fortran programs to be able to access GM primitives. In particular, among other secondary functionality, it provides:

- An initialization function that opens the GM port, parses the machine file to find the peers and sets a global variable to the number of available send tokens. In addition, it initializes some control message queues which are used for communication mechanisms not related to this thesis.
- A finalization function that releases all the resources allocated during initialization and closes the GM port
- A set of functions for regular send and receive, as well as support for control messages, neither of which were used in the final results presented in this thesis
- A simple barrier

- Wrappers for useful C functions such as `gettimeofday()` and `memcpy()`
- A `send address()` and `recv address()` function to hide from Fortran (and the programmer) the exchange of remote pointers
- A function that implements one-sided send, by calling `gm_directed_send()`. The method calls the GM primitive directly without performing any copying or buffer manipulation. Actually in the case where there are no Send Tokens left, the method will wait until one is available before proceeding.

4.3 Experimental Results and Analysis

Figures 4.1-4.3 present the experimental results of magnet. I created a standalone program from the transformed parts of the code, and timed computational kernels only.

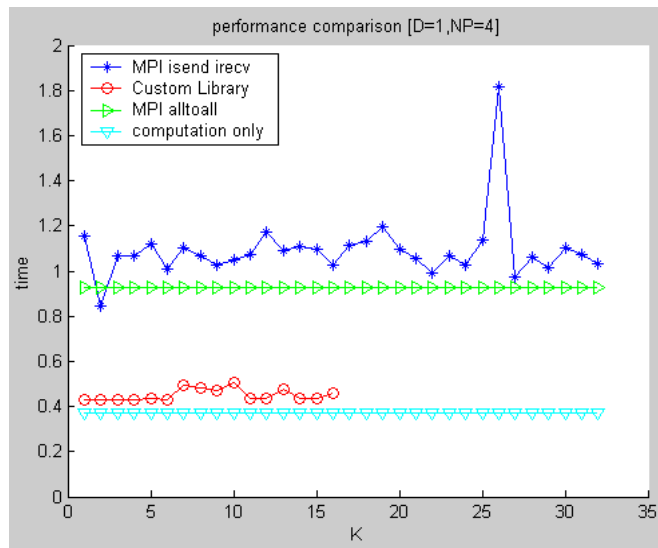


Figure 4.1 magnet's execution time comparison where $D=1$ and $NP=4$ with data $4X512X512$ double precision complex.

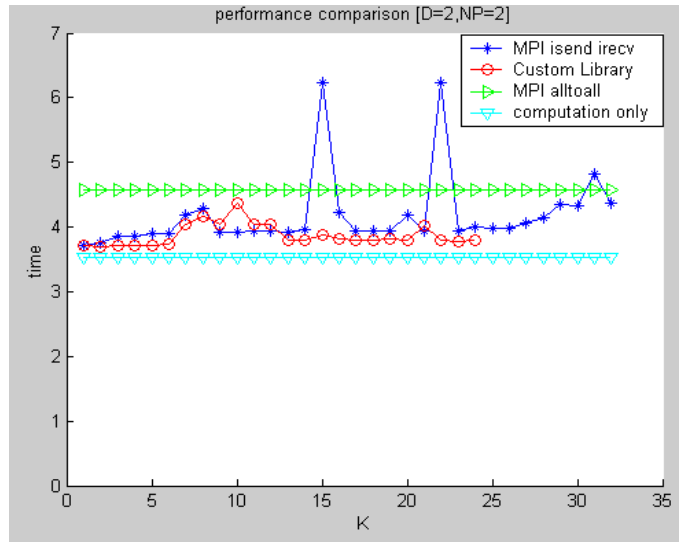


Figure 4.2 magnet's execution time comparison where D=2 and NP=2 with data 4X1024X1024 double precision complex.

Figures 4.1 and 4.2 show execution times for four different communication strategies using MPICH-GM, under different values for K, fixed D=1, and number of processes equal to 4 and 2 respectively. There is some noise (example, case K=26 in Figure 4.1) in the graph. However, there is adequate data to see the general trends of overall performance in each program. The horizontal axis depicts different K values used to execute each version, and the vertical axis represents execution times in seconds. The cyan line represents computational kernel time only, which is the minimum time that can be achieved. The green lines represent the execution times with the MPICH-GM's MPI_ALLTOALL() function used. The blue line represents the execution time with MPICH-GM's MPI_ISEND() function used. The red line represents the execution time with the Custom Library used to utilize one-sided I/O. Due to the limited memory size that can be used with DMA, the Custom Library version could not increase K and D val-

ues as much as the other versions. For example, the MPICH-GM MPI ISEND version used K values upto 32, but the Custom Library could only have K values upto 16 in Figure 4.1-4.2.

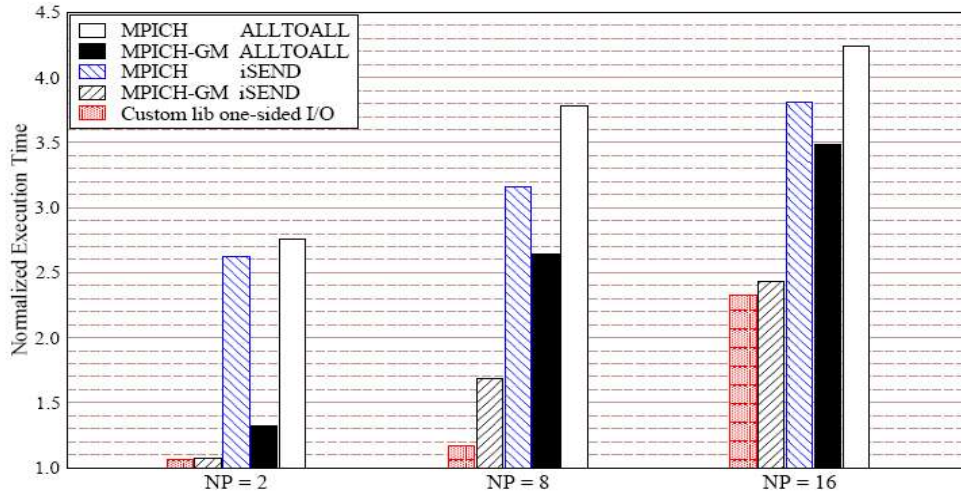


Figure 4.3 *magnet's* normalized execution time comparison for optimized K and D with data 4X1024X1024 double precision complex.

Figure 4.3 displays normalized execution times of each different version of *magnet*. The bars represent the overall best case for each approach. The first observation is that the performance of programs using MPICH-GM is significantly better than the corresponding ones using MPICH. This is an intuitive result, since MPICH-GM is a vendor-specific (Myricom) MPI implementation which is tuned for the specific network infrastructure. A second observation is that moving from a communication model where all the I/O is performed after the entire computation to one where non-blocking I/O is used to overlap communication and computation, yields significant performance improvements. The third observation is that the custom library's performance is slightly better than the version that

uses MPICH-GM.

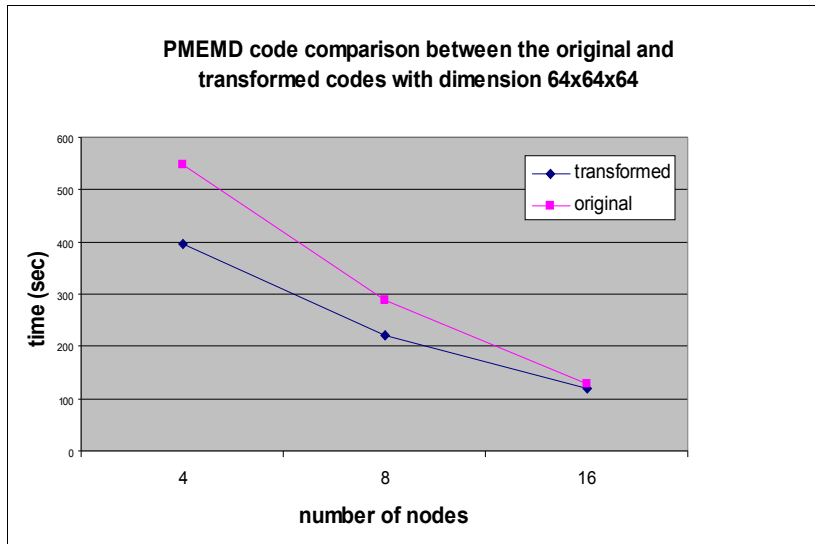


Figure 4.4 PMEMD's execution time comparison with data 64x64x64 double precision.

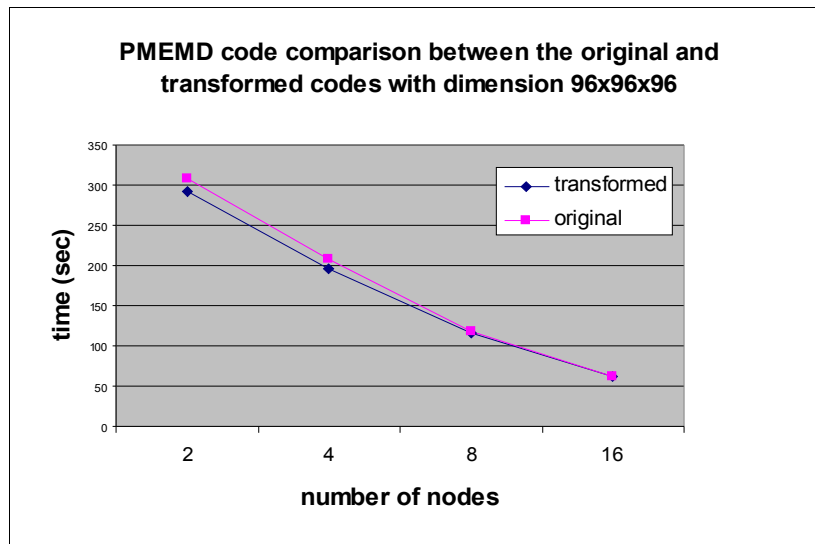


Figure 4.5 PMEMD's execution time comparison with data 96x96x96 double precision.

The figures 4.4-4.5 show the comparison of the original and transformed PMEMD

codes. The original PMEMD version uses `MPI_ISEND()`, `MPI_IRECV()`, and `MPI_WAITALL()` functions. Although `MPI_ISEND()` and `MPI_IRECV()` are nonblocking communication functions, the original PMEMD version does not use pipelining with small communication tiles, so the communication style is equivalent to the ALLTOALL communication with blocking I/O. Since a transformed PMEMD code that uses the Custom Library is under development, I made experiments with only one transformed version that used the MPI's `MPI_ISEND()` which is a non-blocking communication. In the figures 4.4 and 4.5, dimension sizes of 64x64x64 and 96x96x96 are used respectively. There is significant performance improvement where dimension is 64x64x64, figure 4.4, while there is moderate improvement where dimension is 96x96x96, figure 4.5. From both figures 4.4 and 4.5, we can observe that transformation becomes less effective as the dimension size grows and as the number of nodes grows. The reason that the transformed PMEMD code does not scale better than the original code is that I could not increase the value `K` as much as optimal. Therefore, the transformed PMEMD code can not have optimal tile size as the number of nodes increases.

Overall, the transformation to overlap communication-computation improves both the *magnet* and PMEMD codes' performance. Although the transformation with the PMEMD code did not scale as much as desired, the performance gains were significant for most of the cases. Also, I could confirm that a vendor specific MPI implementation (MPICH-GM) can take advantages of the network architecture so that the utilization of the network coprocessors is very high. As we can see from the PMEMD cases, finding the optimal message tile sizes (`K` value) is a crucial factor in the transformed codes' performance.

Chapter 5

RELATED WORK

Many compiler or language-based techniques translate higher-level parallel constructs into message passing primitives as appropriate. Examples include HPF [14], Fortran-D [15], Split-C [16], and more recently UPC [17]. While these approaches allow programs to be written in a single program, multiple data (SPMD) style, they focus on parallel optimization in the large rather than focusing on optimization of messaging on a single host. Due to the difficulties with obtaining satisfactory performance in distributed memory environments, and the lack of ubiquitous availability of parallel languages, many applications are parallelized with explicit message passing calls. These applications are the focus of this work. There have been several research efforts designed to mitigate messaging overheads. These efforts include Active Messages [18], U-Net [19], Fast Messages [20], and the standard produced by Microsoft, Compaq and Intel known as the Virtual Interface Architecture (VIA) [21]. While the performance improvements offered by OS-bypass and user-level networking efforts have been demonstrated, others have observed that these approaches can be difficult to use [21] and that abstractions to make easier use of such techniques can nullify some performance gains [22].

Chapter 6

CONCLUSIONS AND FUTURE WORK

From the experiments performed with *magnet* and PMEMD, I conclude that there is significant performance improvement when the communication-computation overlap transformation is applied with proper values for the critical parameters (K and D). MPICH-GM and the Custom Library we developed also help fully take the advantage of asynchronous communication. By applying the transformation in two different real world applications, we can be confident that the transformation is promising for similar scientific MPI parallel programs.

Although our transformation seems to provide significant performance improvements, finding the optimal values for the parameters K and D is not a trivial task. Our group is currently working on automating this transformation with a source-to-source transformer, called *Compuniformer*. Besides the source-to-source compiler, we are also developing an empirical performance prediction analyzer which will predict optimization parameters, such as K and D, for the source-to-source compiler.

Chapter 7

REFERENCES

- [1] MPI <http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.1/mpi-report.html>
- [2] MPICH <http://www-unix.mcs.anl.gov/mpi/mpich/>
- [3] MPICH-GM <http://www.myri.com/scs/>
- [4] LAM/MPI <http://www.lam-mpi.org/>
- [5] Gm reference manual <http://www.myri.com/scs/GM/doc/refman.pdf>.
- [6] Danalis, Kim, Pollock, and Swany. Transformations to Parallel Codes for Communication-Computation Overlap. Technical report, University of Delaware. 2005. <http://www.cis.udel.edu/~hiper/passages/papers/danalis.tech.05.pdf>
- [7] Overlapping Computations, Communications and I/O in parallel Sorting. Journal of Parallel and Distributed Computing, 28(2):162–172, 1995.12
- [8] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In First International Workshop on Parallel Processing, 1994.
- [9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. IEEE Micro, 15(1): 29–36, 1995.
- [10] L. Bouge, J. Mehaut, R. Namyst, and L. Prylli. Using VIA to build distributed, multithreaded runtime systems: a case study. Research Report 1999-27, CNRS-INRIA-ENS LYON, 1999.
- [11] R. Brightwell and K. D. Underwood. An analysis of the impact of MPI overlap and independent progress. In Proceedings of the 18th annual international conference on Supercomputing, pages 298–305. ACM Press, 2004.
- [12] K. W. Cameron and R. Ge. Predicting and Evaluating Distributed Communication Performance. In Supercomputing, Pittsburgh, PA, 2004.
- [13] F. Chaussumier, F. Desprez, and L. Prylli. Asynchronous Communications in MPI – the BIP/Myrinet Approach. In Euro PVM/MPI '99, 1999.
- [14] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. CRPC-TR92225, Rice University, Houston, TX, 1993.
- [15] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for fortran d on MIMD distributed-memory machines. In Supercomputing, pages 86–100, 1991.
- [16] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in split-c. In Supercomputing, pages 262–273, 1993.
- [17] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. Upc specification v. 1.1.

<http://upc.gwu.edu/documentation>, 2003.

- [18] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In Proceedings of the International Symposium on Computer Architecture, 1992.
- [19] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 40–53, Dec 1995.
- [20] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In Proceedings of Supercomputing '95, 1995.
- [21] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. IEEE Micro, pages 66–76, March/April, 1998.
- [22] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local-area communication with fast sockets. In Proceedings of Usenix Annual Technical Conference, pages 257–274, 1997.