IMPLEMENTATION OF A PERFORMANCE INSTRUMENTATION FRAMEWORK FOR GLOBAL ARRAYS

A Thesis
Presented in Partial Fulfillment of the Requirements for
the Degree of Master of Science in the
Graduate School of the Ohio State University

By
Golam H. Kawsar

The Ohio State University
2008

Master’s Examination Committee:                        Approved By
Dr. P. Sadayappan, advisor
Dr. Atanas Rountev

__________________________
Advisor
Graduate Program in
Computer Science and Engineering
ABSTRACT

The Global Arrays (GA) toolkit implements a shared-memory programming model for distributed memory machines where data locality is managed by the programmer. GA uses an internal one-sided communication engine called Aggregate Remote Memory Copy Interface (ARMCI). ARMCI is a portable, general, and GA-independent communication library that offers both blocking and non-blocking modes of communications. We have implemented a performance instrumentation framework for ARMCI that will generate computation and communication overlap data for applications using the ARMCI library. The performance data will be useful for parallel application developers using ARMCI non-blocking communications routines. We developed a sample application to show the effectiveness of the instrumentation implementation in ARMCI.
I am deeply indebted to my advisor Professor Sadayappan for providing me the opportunity to pursue graduate study and research in his group. I also thank Dr. Jarek Nieplocha at the Pacific Northwest National Laboratory for giving me the opportunity to work there in the summer of 2006 and work on Global Arrays with his team.

I have been helped by the CSE staff and faculty members at OSU in many ways. Their prompt help and sincere cooperation have made my stay at OSU much smooth.

Finally I am so thankful to my friends and colleagues in the CSE department. I felt like at home all the time during my one and half years of stay at OSU just because of the warm and friendly welcome by them. I will always cherish the memories I made here with you guys.
VITA

October 27, 1979 .............................................. Born – Noakhali, Bangladesh

  Shahjalal University of Science and Technology, Bangladesh

  Southeast University, Bangladesh

June 2006 – December 2007 ........................ Graduate Research Associate,
  Ohio State University

PUBLICATIONS

G. Kawsar, S. Chowdhury, A Novel Most Recent Performance – Based Strategy for Load
Balancing in Parallel Computing, in proceedings of ICCIT 2005, Dhaka, Bangladesh,
December 28-30, 2005

FIELDS OF STUDY

Major Field: Computer Science and Engineering
TABLE OF CONTENTS

Abstract .................................................................................................................. ii
Acknowledgement ................................................................................................... iii
Vita .......................................................................................................................... iv
List of Figures ......................................................................................................... vii

Chapters:

1. Introduction ........................................................................................................ 1
   1.1 Background ..................................................................................................... 1
   1.2 Related Work ................................................................................................ 3

2. Non-blocking communications in ARMCI ...................................................... 6
   2.1 ARMCI Data Transfer Routines Background ............................................. 6
   2.2 Non-blocking Data Transfer Mode ............................................................. 7

3. The Overlap Measurement Framework ......................................................... 12
   3.1 Framework Parameters ............................................................................... 12
   3.2 Overlap Measurement Technique ............................................................. 15

4. Implementation of the Instrumentation Framework in ARMCI ............... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 The High Level Implementation Approach</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Implementation Details</td>
<td>19</td>
</tr>
<tr>
<td>5. Experimental Results</td>
<td>23</td>
</tr>
<tr>
<td>5.1 Test Environment</td>
<td>23</td>
</tr>
<tr>
<td>5.2 Test Results</td>
<td>23</td>
</tr>
<tr>
<td>5.3 Test Results Analysis</td>
<td>27</td>
</tr>
<tr>
<td>6. Conclusion and Future Directions</td>
<td>29</td>
</tr>
<tr>
<td>Bibliography</td>
<td>30</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Deciding to use zero-copy or not</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>ARMCI non-blocking routines</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Operation of a non-blocking routine in ARMCI</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Important events in non-blocking data transfer</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Relationship between instrumentation layer and other components</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Process flow in collecting event data</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Overlap data table for data size of 32 MB</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>Overlap data chart for data size of 32 MB</td>
<td>25</td>
</tr>
<tr>
<td>5.3</td>
<td>Overlap data table for data size of 64 MB</td>
<td>25</td>
</tr>
<tr>
<td>5.4</td>
<td>Overlap data chart for data size of 64 MB</td>
<td>26</td>
</tr>
<tr>
<td>5.5</td>
<td>Overlap data table for data size of 128 MB</td>
<td>26</td>
</tr>
<tr>
<td>5.6</td>
<td>Overlap data chart for data size of 128 MB</td>
<td>27</td>
</tr>
<tr>
<td>5.7</td>
<td>Sample profiling data</td>
<td>29</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Background

With widening gap between processing power and network I/O in parallel computers, there have been developed many techniques to utilize idle CPU cycles while data is being transferred in the network. One such technique is the use of asynchronous communications between nodes in parallel computers. Asynchronous communications can be utilized by the concept of computation-communication overlap in which computation and communications are performed in parallel. This is achieved by Network Interface Cards (NIC) that can access system memory without processor involvement. Modern networks like Infiniband [4], Myrinet [3], and Quadrics [5] all have supports for asynchronous data transfer.

Global Arrays (GA) is a parallel programming tool that was created to provide application programmers with an interface that allows them to distribute data while maintaining the type of global index space and programming syntax similar to what is available when programming on a single processor. The goal of GA is to free the programmer of the burden of managing explicit communications between processors by making calls to functions that transfer data between global address space to local storage.
Aggregate Remote Memory Copy Interface (ARMCI) [2] is a portable remote memory copy library that is part of the Global Arrays (GA) toolkit. GA uses ARMCI as the primary communication layer. Neither GA nor ARMCI can work without a message-passing library that provides the essential services and elements of the execution environment (job control, process creation, interaction with the resource manager). GA inherits the Single Program Multiple Data (SPMD) model of computations from MPI, along with the overall execution environment and services provided by the operating system to the MPI programs.

In addition to being the underlying communication interface for Global Arrays, ARMCI has been used to implement communication libraries and compilers [6][7][8]. ARMCI offers an extensive set of functionalities in the area of RMA communication:

1) data transfer operations, 2) atomic operations, 3) memory management and synchronization operations, and 4) locks. Communication in most of the non-collective GA operations is implemented as one or more ARMCI communication operations.

ARMCI was designed to be a general, portable, and efficient one-sided communication interface that is able to achieve high performance [9][10][11]. It also avoided the complexity of the progress rules and increased synchronization in the MPI-2 one-sided model that contributed to its delayed implementations and still rather limited adoption.

ARMCI uses one-sided communication mechanism in which the receiving node does not need to get involved in the data transfer process. This is achieved by using Remote Direct Memory Access (RDMA) technology. RDMA is different from the traditional “two-sided” Send/Receive model in that in the Send/Receive model, the source issues a Send
that describes the location of the data to be sent, the destination posts a Receive that similarly indicates where the data is going to be written, and a matching capability is used to associate a posted Receive to an incoming Send. Each side has part of the information required for the completion of the communication; this is a "two-sided" interface. With RDMA, on the other hand, both origin and destination buffers must be registered prior to any operations. This memory registration returns a handle that can be used in RDMA operations, Read or Write (a.k.a. Get or Put), to describe the origin or destination buffer. Only one side needs to have all of the information required for the completion of the communication; this is a "one-sided" interface.

ARMCI does not have an overlap and overhead measurement mechanism built in. We have implemented an instrumentation framework for ARMCI. The instrumentation framework is similar to the one proposed by Aniruddha Shet, et al. [1]. This is a general purpose instrumentation framework applicable to message-passing parallel programming models like MPI that have non-blocking data transfer routines.

1.2 Related Work

There are quite a few tools available for performance analysis of parallel programs. The most prevalent approach taken by these tools is to collect performance data during program execution and then provide post-mortem analysis and display of performance information [16]. Some tools do both steps in an integrated manner, while other tools or tool components provide just one of these functions. SvPablo [17] is a language independent performance analysis and visualization system that supports application
performance analysis. It can insert instrumentation code automatically and provides performance data for numerous metrics. TAU (Tuning and Analysis Utilities) [18] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, Python. TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements. COMB [19] is a portable benchmark suite that assesses the ability of cluster networking hardware and software to overlap MPI communication and computation. COMB measures the relationship between overall MPI communication bandwidth and host CPU availability. PERUSE [15] is a performance-revealing extensions interface to MPI. This interface is intended to provide greater insight into the performance related processes and interactions between application software, system software, and MPI message-passing middleware than the standard MPI Profiling interface (PMPI) defined by the MPI specification. The Paradyn [20] tool leverages a technique called dynamic instrumentation to obtain performance profiles of unmodified executables.

Almost all of the above tools are trace-based and generate huge amount of profile data, which is hard to manage and manipulate. Since overlap measurement is our only goal, our instrumentation implements a much simpler method of overlap measurement without necessarily generating a lot of trace data. Thus, if overlap measurement is the only goal, our implementation does that with minimal effort.

The rest of this thesis is organized as follows:
Chapter 2 goes through the non-blocking capabilities of the ARMCI library, chapter 3 describes the instrumentation framework that we are going to implement, and chapter 4 describes the implementation of the framework in ARMCI. In chapter 5 we detail the experiments and test results on our implementation. Chapter 6 concludes the thesis with some final remarks and future directions along performance instrumentation research.
CHAPTER 2

NON-BLOCKING COMMUNICATIONS IN ARMCI

2.1 ARMCI Data Transfer Routines Background

Modern scientific, engineering, and business computing applications are data and computation hungry. Such applications often require transfers of noncontiguous data that corresponds to fragments of multidimensional arrays, sparse matrices, or other more complex data structures. Applications running on multi-node clusters need to move data between nodes reliably and quickly. With remote memory communication APIs that support only contiguous data transfers, it is necessary to transfer noncontiguous data using multiple communication operations. This often leads to inefficient network utilization and involves increased overhead. ARMCI offers explicit noncontiguous data interfaces in two formats - strided and generalized I/O vector that allow description of the data layout so that it could, in principle, be transferred in a single message. The effectiveness of actual transfers depends on the ability of the underlying network to deal with noncontiguous data (e.g., scatter/gather operations). However, even when scatter/gather operations are not supported by the network, the ARMCI strided and vector operations take advantage of the information, so that the overall number of messages and network packets transferred is reduced. Although the explicit message aggregation accomplished through the use of strided and vector interfaces is an effective mechanism
for reducing communication overhead, it does not exploit all the available opportunities or optimization. ARMCI employs techniques such as non-blocking Remote Memory Copy (RMA) and implicit communication aggregation that help for latency tolerance [13].

2.2 Non-blocking Data Transfer Mode

Non-blocking Data transfers initiate a communication call and then return control to the application. Because the RMA model is simpler than MPI (e.g., does not involve message tag matching or dealing with early arrival of messages), in principle more opportunities for overlapping communication with computation are available. However, these opportunities are not automatically exploited by deriving implementations of non-blocking APIs from their blocking counterparts. For example, the communication protocols used to optimize blocking transfers of data from non-registered memory by pipelined copy and network communication through a set of registered memory buffers [10] can achieve very good performance by tuning the message fragmentation in the pipeline [14]. But the memory copy requires active host CPU involvement and therefore reduces the potential for effective overlapping communication with computation. To increase the overlap, ARMCI expanded the use of direct (zero-copy) protocols on networks that require memory registration, such as Myrinet. In ARMCI, a return from a non-blocking operation call indicates a mere initiation of the data transfer process, and the operation can be completed locally by making a call to the wait routine. Waiting on a non-blocking put or an accumulate operation ensures that data was injected into the network and the user buffer can now be reused. Completing a get operation ensures that
data has arrived into the user memory and is ready for use. A wait operation ensures only local completion. The library imposes a limit on the number of outstanding requests allowed (if necessary, it can transparently complete an old request and free up the resources for a new request).

Data transfers can be either in zero-copy or copy mode. In the zero-copy mode, if the underlying network allows, data is transferred directly from the user buffer to the destination buffer in the remote machine. For example, choosing copy/zero-copy policy on Infiniband networks can be done with the following techniques:

a. dynamic registration/de-registration,

b. copy using previously registered memory, and

c. using a memory allocation interface.

A technique from the above 3 is chosen based on following flowchart:

Can the memory containing data be registered (does memory range fit into the table?)?

no

Depending on the size of the message, either use pre-allocated registered memory, or use a memory allocation interface

yes

Use Infiniband zero-copy RDMA Read/Write protocol

Figure 2.1: Flowchart of deciding to use zero-copy or not
The second technique, i.e. using pre-allocated registered memory is further improved by pipelining the memory copy and data sending process. The data memory is divided into segments, and segments are copied and sent injected into the network in parallel in a pipelined fashion. This hides latency to a great extent.

Unlike their blocking counterparts, the non-blocking operations in ARMCI are not ordered with respect to the destination. Other than performance, another reason is that by ensuring ordering, the library incurs additional and possibly unnecessary overhead on applications that do not require ordered operations. When necessary, ordering can be done by calling a fence operation. The fence operation is provided to the user to confirm remote completion if needed.

<table>
<thead>
<tr>
<th>Non-blocking routines</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMCI_NbGet, ARMCI_NbGetS, ARMCI_NbGetV</td>
<td>Contiguous, vector, and strided versions of get</td>
</tr>
<tr>
<td>ARMCI_NbPut, ARMCI_NbPutS, ARMCI_NbPutV</td>
<td>Contiguous, vector, and strided versions of put</td>
</tr>
<tr>
<td>ARMCI_NbAcc, ARMCI_NbAccV, ARMCI_NbAccS</td>
<td>Atomically updates memory of remote of a remote machine</td>
</tr>
</tbody>
</table>

Figure 2.2: ARMCI non-blocking routines
These non-blocking data transfer routines allow users to write applications that take advantage of the overlap of computation with communications. Our work involved instrumenting the ARMCI library so that it gives a summary of the actual overlap achieved and overhead incurred during the execution of an application. This information is useful for the parallel application developers because it pinpoints the areas where the application can make improvement in terms of execution times by latency hiding and taking advantages of idle CPU cycles.

A typical non-blocking routine is implemented in the following manner:

```
ARMCI_NbOp(src/dest, info, handle)
```

Figure 2.3: Operation of a non-blocking routine in ARMCI
Figure 2.3 shows how a non-blocking routine is processed in ARMCI. The non-blocking model in ARMCI was implemented with the following goals in mind: the functionality must be implementable across a wide variety of platforms, performance advantages of the native communication protocols must be exploited, opportunities for overlapping communication and computations should be provided, and as much of the code as possible must be shared to minimize the maintenance efforts across different platforms [21]. On networks like the IBM SP interconnect and Quadrics, the underlying RMA layer provides most of the required capabilities. Hence, on these systems, most of the non-blocking calls can be implemented as thin wrappers to the native protocols. These protocols are referred to as direct protocols. Some networks like Infiniband and Myrinet use a zero-copy strategy while some (like IBM LAPI) have the native communication interface copy data internally. Some networks like GM, VIA, and InfiniBand require data to be transmitted from/to special memory [21]. This can be accomplished either by 1) copying the data into a set of special registered/pinned buffers for transmission; 2) allocating registered memory for the user; or 3) by on-demand registration of the user’s memory. ARMCI uses all three schemes, depending on the platform, operation type, or size of the data transfer. Protocols that use memory copy scheme are referred to as buffered. Figure 2.3 illustrates these protocols processing in ARMCI.
3.1 Framework Parameters

The instrumentation we used follow MPI PERUSE’s [15] specifications of internal library events at different points of the data transfer process. For our overlap measurement process, the following events are used:

CALL_BEGIN:

This is the event when execution inside a data transfer routine begins.

CALL_END:

Event when execution leaves a data transfer routine.

TRANSFER_BEGIN:

Event when data transfer actually begins. This happens some time after the CALL_ENTER event, but before the CALL_EXIT event.

TRANSFER_END:

Event when data transfer actually ends. Like the TRANSFER_BEGIN event, this too will happen after the CALL_ENTER event. Also, it will typically happen after
the CALL_EXIT event. But this is not guaranteed – for small data transfers, the data transfer may complete before control is returned back to the data transfer routine.

**WAIT_BEGIN:**

Event when a wait routine is called. This event marks the beginning of the wait for completion of data transfer.

![Diagram of Important Events in Non-blocking Data Transfers](image)

**Figure 3.1:** Important events in non-blocking data transfers
WAIT_END:

Event when a wait routine returns. This event takes place only when the wait routine determines that the data transfer is complete.

Figure 3.1 illustrates the time events at different points of the overall data transfer process. It shows two processors, processor 0 and processor 1, possibly on different nodes on a parallel computer, are transferring data using the ARMCI_NbPut non-blocking routine. We can break down the phases in a non-blocking transfer as follows:

1. Time inside the data transfer routine:

   This is the time spent between CALL_BEGIN and CALL_END of the data transfer routine. Data transfer is initiated during this time. Although in most cases, the actual physical data transfer will be completed after this time is over, there can be cases when data transfer also completes within this period. This can happen, among other reasons, when the data to be transferred is small and the network is relatively fast.

2. Time between CALL_END and WAIT_BEGIN:

   This is the time period of most interests to us. This is where the potential overlap may take place. As the execution proceeds past CALL_END, the data transfer is already initiated, and the RDMA engine copies data from the source to the destination without host processor involvement in either node. With one-sided mechanism, the receiving processor is also not involved in the transfer process.
Any computation during this phase that is executed in parallel to the actual, physical data transfer is a pure gain in performance as the relatively slower network I/O is not bottlenecking the processor, which is now free to do other necessary computation the application needs to do.

3. Time between WAIT_BEGIN and WAIT_END:

This is the time that is passed on purely waiting for the data transfer to complete. The ARMCI_Wait routine checks to see if the data transfer is complete and returns control to the user application if it is complete. If data transfer is still underway, the wait routine blocks until data transfer is not complete. There cannot be any computation during this time. From a performance point of view, this period of time is a pure waste, as it a blocking call and no useful computation can be done during this phase.

3.2 Overlap Measurement Technique

Overlap of computation and communication is measured in the following way:

We instrument the entire AMRCI library to timestamp the events we described before. Then we add up all the times between the CALL_END and WAIT_BEGIN events of all the non-blocking routine and their corresponding wait call invocations. We also calculate the total amount of data transferred by the non-blocking routines from the application. We need one last parameter before we can measure the overlap. This last parameter can be named Data_Transfer_Time. Data Transfer Time is the time needed for the data to be
transferred from source to destination. This time can be measured independent of the process of time-stamping the other events needed for overlap measurement. With all these, overlap can then be measured with respect to the following scenarios:

1. TRANSFER_END event occurred before CALL_END event. Potential overlap is this case is clearly none. This is because the data transfer is complete before the execution is returned back to the application. There can be some level of computation going in parallel with the data transfer only if there are more data to be transferred during when the application is doing some computation. Since all data have been transferred already, there is no overlap in this case.

2. TRANSFER_END event occurred after CALL_END but before WAIT_ENTER. In this case there can be some level of overlap. The potential overlap period is the time between the events CALL_END and TRANSFER_END. The amount of overlap depends on when the TRANSFER_END event occurred. If 

   \[ \text{Data\_Transfer\_Time} \leq \text{time between CALL\_END and WAIT\_BEGIN} \]

   then the maximum possible overlap can take place. Otherwise the amount of overlap is less than the maximum achievable overlap. This is because when 

   \[ \text{Data\_Transfer\_Time} \leq \text{time between CALL\_END and WAIT\_BEGIN} \]

   we can use this entire period of time (between CALL_END and WAIT_BEGIN events) to do useful computation. Since this time is the \( \geq \) Data_Transfer_Time, this is the maximum possible overlap of computation with communication.
3. TRANSFER_END event occurred after WAIT_BEGIN and before WAIT_END. 
   This is effectively the same case as the previous one. Here too the amount of 
   potential overlap is limited by the time period between CALL_END and 
   WAIT_BEGIN. Depending on whether this time period is greater or smaller than 
   the Data_Transfer_Time, there can be maximum or less than maximum 
   achievable overlap.

   The amount of overlap determined using the above approach is not precisely accurate. 
   This is because we are unable to timestamp the actual data transfer event in the network. 
   The actual time when physical data transfer begins (after the ARMCI library has issued a 
   transfer request) is decided by the underlying network hardware and we don’t currently 
   have a mechanism to get the exact time when the underlying network begins the physical 
   data transfer. Still, we get a very good idea of the amount of overlap following the above 
   approach.
CHAPTER 4

IMPLEMENTATION OF THE INSTRUMENTATION FRAMEWORK IN ARMCI

4.1 The High Level Implementation Approach

The basic strategy to implementing the overlap measurement framework is to timestamp the events as they happen, and then store and analyze the events to get the overlap and overhead of the application in using ARMCI.

The entire implementation thus lies completely inside the ARMCI library itself. This makes the implementation highly portable as the library itself. The instrumentation framework does not need to rely on knowledge of the application being tested.

Figure 4.1 shows the high level implementation structure of the framework. The instrumentation layer resides inside the library which functions on top of the lower level OS and network-specific services. The library runs on a variety of platforms and networks, which does not affect our instrumentation framework in any way.

The user program does not need any knowledge about the instrumentation layer, and similarly the instrumentation does not need any knowledge of the user program.
4.2 Implementation Details

The implementation was done in two parts. The first part creates the call structures and fills them as the events occur. After registering the call events, it saves the event data into a file on disk. The second part is the data analyzer part, which basically goes through the stored data and calculates the amount of overlap achieved and overhead incurred by the application.
The primary structure object we used to capture the events in the data transfer process is as follows:

```c
typedef struct
{
    int call_id;
    int proc_id;
    double call_enter;
    double call_exit;
    double wait_enter;
    double wait_exit;
    int bytes_transferred;
    int isblocking;
    int iswait;
    armci_hdl_t *handle;
} call_data_t;
```

This structure object contains all variables necessary to store data associated with an event. The variable names are self-explanatory. We record the times of call entry, call exit, wait routine entry, wait routine exit, number of bytes being transferred, whether the call is a blocking or non-blocking call, and the handle, if it is a non-blocking call.

The data are collected on a per-call basis. After the application finishes execution, we have an event data file that contains the timestamps for all the events and also other associated data. Data are written out to the file whenever the in-memory buffer is full. The instrumentation can be expanded in future to add new features by adding new data members to the structure object.
Our data analyzer goes through the file that recorded all the events and calculates the overlap achieved and overhead incurred during the execution of the application. The measurement approach follows the event model described in the previous chapter.

Figure 4.2 shows a schematic diagram of the process flow in the instrumentation process.

Figure 4.2: Process flow in collecting event data
As events are triggered from the user application, they are captured and registered in an internal storage. The internal storage is emptied out to a disk file when it becomes full. The file in disk is the final storage of all the events data that are later analyzed to determine the degree of overlap in the user program.

Since all data recorded are on per-call basis and we determine the overlap amount application-wide, we aggregate each call’s overlap contribution to the total, application-wide overlap measurement. Computation times are added to find out total computation through the execution of the program.
CHAPTER 5
EXPERIMENTAL RESULTS

5.1 Test Environment
We ran our tests on the OSC Itanium Cluster. It has two partitions of dual processor nodes connected with 2 Gbit/s Myrinet high speed interconnects which are available for parallel applications using MPI for message passing. One partition is of 128 nodes with two 900MHz processors and 4GB of memory per node, and the second partition is of 110 nodes with two 1.3 GHz processors and 4GB of memory per node. The cluster runs on the Linux operating system.

5.2 Test Results
We used a small microbenchmark to run our overlap measurement experiment. The microbenchmark simply transfers some data from one node to another using ARMCI’s non-blocking routines.

In the process of generating overlap and overhead information, our implementation also generates a host of other information about the application run. We generate the number of times each data transfer routine was called, whether it is a blocking or a non-blocking call, how much data was transferred, etc. All these data give an overall idea of the
application, which can be analyzed by the application developer to find any performance hole in the application.

The test program we have written uses the non-blocking ARMCI_NbPut routine to transfer different amount of data from one processor in one to another processor on a remote node.

We have run our test application for different amount of data transfers and computation times. The following tables show how overlap percentages change with varying computation and communication times.

<table>
<thead>
<tr>
<th>Data Transferred(MB)</th>
<th>Computation Time (milliseconds)</th>
<th>Overlap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>59.236</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>77.666</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>154.17</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>308.540</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>442.12</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>498.79</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 5.1: Overlap data table for 32 MB of data transfer

The chart that results from the above table (the computation time has been rounded a bit for clarity):
Figure: 5.2: Overlap data chart for 32 MB of data transfer

The same test for 64 MB and 128 MB of data follow:

<table>
<thead>
<tr>
<th>Data Transferred(MB)</th>
<th>Computation Time (milliseconds)</th>
<th>Overlap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>77.580</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>142.65</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>194.56</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>285.62</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>462.67</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>616.57</td>
<td>100</td>
</tr>
</tbody>
</table>

Fig 5.3: Overlap data table for 64 MB of data transfer

A chart showing the data of the above table (Computation time rounded a bit for clarity):
Fig 5.4: Overlap data chart for 64 MB of data transfer

<table>
<thead>
<tr>
<th>Data Transferred (MB)</th>
<th>Computation Time (milliseconds)</th>
<th>Overlap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>25.655</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>51.386</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>102.76</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>205.90</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>257.87</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>385.52</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>513.89</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>646.16</td>
<td>100</td>
</tr>
</tbody>
</table>

Fig 5.5: Overlap data table for 128 MB of data transfer

A chart for the above table is (Computation time rounded a bit for clarity):
Looking at the tables and charts above, we can see how amount of overlap is affected with respect to amount of data transferred and amount of computation inserted between a non-blocking call and its corresponding wait function. With small data transfers, we can get a high amount of overlap with a small amount of computation. But with a large amount of data transfer, we must also have more computation to fill in the period while non-blocking transfer will be going on in parallel.

For example, when we are transferring 32 MB of data, we get 100% overlap with just 154 milliseconds of computation inserted in between a call to a non-blocking routine and its corresponding Wait call, but with only 154 milliseconds of computation while transferring 128 MB of data, we get about 25% overlap.
Referring to our terminology, to get maximum possible overlap (100%), the computation inserted between a CALL_END and WAIT_BEGIN should be greater than or equal to the Data_Transfer_Time of the data being transferred. If we have less computation than the Data_Transfer_Time then we will be wasting time in the Wait routine, which is a completely idle time for the CPU. To maximize performance by hiding latency under computation, parallel application developers using non-blocking routines should always insert as much computation as possible before calling the Wait routine. Ideally, there should not be any wait inside the Wait routine!
CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

In this thesis, we described our implementation of an instrumentation framework for the ARMCI communication library. We showed how this instrumentation can reveal overlap information about any application using the ARMCI as its communications layer. The sample application we wrote to find the LU decomposition of large matrices shows the effectiveness of the framework implementation.

Going forward, we can add the overhead measurement module in the framework. The overhead number will give an idea of the amount of time the execution is confined inside the ARMCI library itself. This information, combined with the overlap data, will be useful in gauging the efficiency of the application as well as the ARMCI as a communication library. Overlap performance numbers obtained from this instrumentation framework will be valuable to application writers in finding out parts in their code that can be modified to improve performance by increasing the amount of overlap.
BIBLIOGRAPHY


