Analyzing and Optimizing Global Array Toolkit for Cray Gemini Interconnect

Vairavan Murugappan

August 27, 2010

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2010

Abstract

Partitioned Global Address Space (PGAS) languages are the newly approaching trend in the High Performance Computing, due to its ease of programming and increasing efficiency of the one sided communications. But without the support of Global addressing (or) one sided communication in hardware it is always going to perform worse than the standard Message Passing Interface (MPI) programs. But with the introduction of Global Addressing on the hardware in the Cray's new Gemini Interconnect, the trend is set to change. With the ease of programming and fully supported hardware, Global addressing based programming will catch the attention of many programmers.

Global Array Toolkit provides the efficient and portable "shared memory" programming interface for distributed –memory systems. It is present even before the PGAS languages where introduced and this forms the basis of the most widely used computational chemistry package NWChem. With the performance which is already very close to the standard Message Passing Interface, this is sure to get benefitted with the introduction of new Gemini Interconnect.

So this forms the main aim of this project, which is to analyze and optimize this Global Array Toolkit, for the Cray XT Systems with Seastar and Gemini Interconnect. A General Benchmarking of the Toolkit is followed by detailed analysis of different optimization strategies for bringing out a better Performance from this library is done.

Contents

List of Tables

List of Figures

List of Graphs

Acknowledgements

I would like express my sincere gratitude to my supervisor Dr. Andrew Turner, EPCC for his advice and supervision throughout the duration of this dissertation. His continuous support and patience added much more to my graduate experience.

I would also like to thank my co supervisor Dr. Jason Beech-Brandt, Cray centre of excellence for his support and help during the initial phase of the project and also for providing me with valuable information from Cray's technical team.

I must also acknowledge Ryan Olson of Cray for his suggestions and help in understanding the implementation of ARMCI.

Finally I would like to thank my parents. Without their support and assistance, I would not have been able to pursue this course.

Chapter 1

Introduction

Supercomputers built from commodity based computers differ from the normal computers by the speed and efficiency of their network (Interconnects). If many processors are connected together using a low speed network it does not make any sense in a field which requires high performance. So Interconnects forms the heart of all massively parallel systems.

The new Gemini Interconnect has many better features than Seastar. With the current trend in high performance computing languages shifting towards the Global Addressing and one sided-communication languages, the effective architectural support for global addressing in Gemini interconnect is very encouraging. Even though Seastar Interconnect seems to provide one-sided communication operation, in truth they are two sided at network level and appears to be one-sided at user level. So the performance of Gemini interconnect, which is one-sided even on the network level is expected to be better than the Seastar chip.

ARMCI[1], the one-sided communication library used by Global Array Toolkit can be used on many numbers of platforms apart from these Cray systems as well. In this project we are only considering the Portals and Gemini implementation of this communication library. This project is carried out on a Cray XT System HECToR, (UK's National Supercomputer). This system has Seastar as its Interconnect. All the results and graphs shown in this report are all done in it.

Cray XT System code named Piz Palu which is installed on CSCS, the Swiss National Supercomputing Computing Centre has Gemini Interconnect in it. This system is used for analysing and optimizing the performance of the Global Array Toolkit on Gemini Interconnect. But due to the agreement with The Cray Supercomputing Company, these

[1] Aggregate Remote Memory Copy Interface

results cannot be published. But all the optimizations analysed and proposed in this project holds well for both these Interconnects.

Chapter 2, explains the concepts and various architectures used in this project. This gives the reader a better understanding of the environment in which all the work has been done in this project.

Chapter 3, describes the initial benchmarking done on the toolkit with various codes and gives an estimate on the current performance of the Toolkit. These values are used as guidelines for comparison throughout the report.

Chapter 4, this explains the different optimization strategies implemented on the toolkit. And also gives the performance variations on the benchmarking codes with previous versions.

Chapter 5, explains the implementation details of the ARMCI operations on the Cray systems and thereby explaining the drawbacks in it, Finally proposing a different type of implementation for these operations.

Chapter 6, gives an overall result of the optimizations done on the toolkit. And also discusses about the performance variations obtained from it.

Conclusions and further work that can be done on the project can be found in the Chapter 7 along with the post-mortem of the dissertation.

Chapter 2

Background Theory

This chapter briefly explains the major concepts referred in this report. It is really not possible to produce all the theories behind the hardware and software layers involved in the project. But care has been taken to make this Chapter as a guidance or reference for this report.

This project is mainly carried out on the UK National supercomputer called HECToR, which is a Cray machine comprising of both XT5h and XT6 machines. The phase2a of HECToR is an XT5h (XT4 and X2 vector machine) machine consisting of 3072 nodes with 4 cores on each node with a total of 12288 cores and phase 2b is an XT6 machine consisting of 24 cores on each of the 1856 nodes. So it is really essential to have good background knowledge about the Cray XT Architecture. This is covered in Section 2.1.1

Both the phases of this system are currently using Seastar 2+ Interconnect, which is expected to be upgraded to Gemini Interconnect at the end of 2010. With Gemini Interconnect providing lots of advanced features than Seastar2+ this seems to be a promising upgrade. Design and working of Seastar2+ and Gemini Interconnects are covered in Sections 2.1.2 and 2.1.3 respectively.

Since this project is mainly about optimizing Global Array Toolkit, it is essential to known what it is and how it works. A brief introduction to GA is given in Section 2.2.1. GA is built on top of a one-sided message passing interface called Aggregate Remote Memory Copy Interface (ARMCI). The bond between GA and ARMCI is implemented very perfectly; hence porting GA to any platform inherently comes down to the task of porting ARMCI, which is covered in Section 2.2.2.

Seastar2+ Interconnect which is currently used in HECToR uses Portals as its Network level API. This is to be replaced with a new Network level API called DMAPP[2] on the

[2] Distributed Shared Memory Application API

new Gemini Interconnect. More information about these API's can be found on Sections 2.2.4.

1.1 Hardware

2.1.1 Cray XT Architecture

Cray XT series systems are commodity based Massively Parallel Systems which can effectively scale from 100 to 250,000 processors. The primary difference among the different series of XT systems (like XT3, XT4, XT5, XT6) are based on their computing node capacities.

2.1.1.1 Basic Hardware overview

Nodes form the logical and processing components of the system. The Cray XT system has 2 types of nodes: Compute Nodes and Service Nodes.

Each Compute Nodes consists of AMD[3] Opteron Processor, DIMM[4] Memory and an Application specific Integrated circuit (ASIC). Number of processors or cores may vary from single, dual to hexa cores based on the series of XT machine with their own local DIMM memory which can be a DDR1, DDR2 or DDR3 depending on the Series. AMD Processor along with its local DIMM memory together forms a NUMA[5] node. These NUMA nodes within a single compute node are connected through a hyper transport link. These entire XT series machine uses Seastar2+ as their ASIC, and the systems which are going to be upgraded to Gemini are code named as XE series.

[3] Advanced micro devices , http://www.amd.com

[4] Dual Inline Memory Module

[5] Non Uniform Memory Access

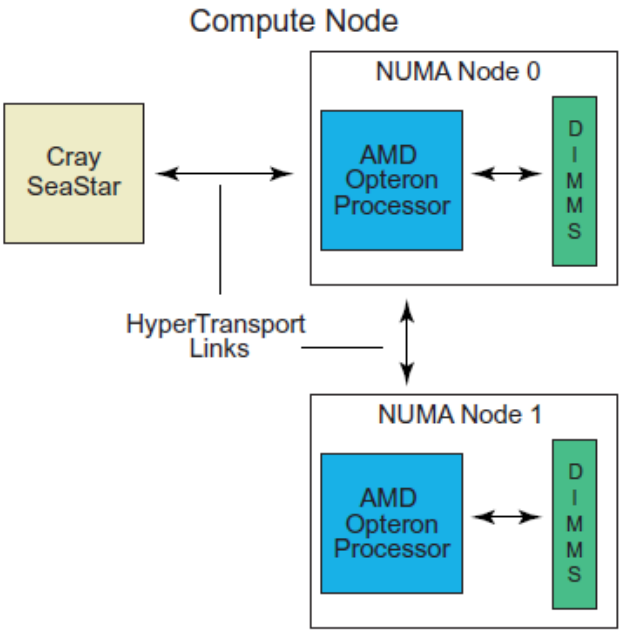


Figure 2-1 Compute Node with 2 Processors (Dual or Quad or Hexa Core)
Image Source : The Cray Supercomputing company (1)

Service nodes in a Cray XT systems consists of a single or dual core processor and a DIMM Memory, a Seastar chip and in addition each service node contains two PCI-X or PCIe slots which are used to connect to different optional interface cards depending on their functionality. They PCI[6] system functions like User login (Login Nodes), Network service nodes for connecting to Network storage devices, I/O Nodes connected to Lustre-manage RAID storage, Service database node to manage system state and Boot Node.

Cray XT system uses x86 based AMD Opteron processors. Number of cores on the processors varies depending on the series of the system. Each processor has two levels of private cache and a third level of shared cache among the cores. First level (L1) cache has 64 KB of data and instruction cache separately. Followed by 512 KB of L2 and 2 MB of L3 cache (values are based on Phase 2a of HECToR (2)).

[6] Peripheral Component Interconnect

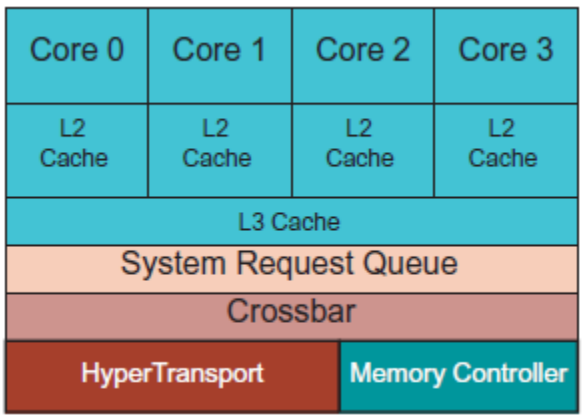


Figure 2-2 Quad Core Processor
Image Source: The Cray Supercomputing Company (1 p. 17)

Cray XT systems use a 3D Torus topological Network, thus allowing all the three dimensions to be cyclic. Each node in a XT machine is connected to the 3D Torus network through an Interconnect.

2.1.1.2 Operating System and File system

Cray's Operating system, Cray Linux Environment is a distributed system of compute node and service node kernel. The Compute Node Kernel (CNL) is a Cray modified light weight implementation of SUSE Linux. This improves the performance of the compute node by eliminating the unwanted features and services. On the other hand, the service nodes run a fully featured version of SUSE Linux.

Lustre is the parallel file system used by Cray XT machines which offers high performance, scalability and POSIX compliances. LUSTRE servers are backed by RAID storage. CNL supports I/O operation to LUSTRE file systems.

2.1.2 Seastar2+ Interconnect

Each Node in a Cray XT machine has its own Seastar2+ (ASIC chip) Interconnect, which connects the processors to the 3D Torus Network. This communication chip takes care of the message passing function of the node, thereby reducing the work load of the main processor. In addition to message passing function of that particular node, it is also involved in routing other messages which passes across it to other nodes. Each Seastar2+ chip has 6 channels (6 Port Routers) each with bandwidth of 9.6 GB/s forming a total theoretical bandwidth of 57.6 GB/s per chip (1 p. 19).

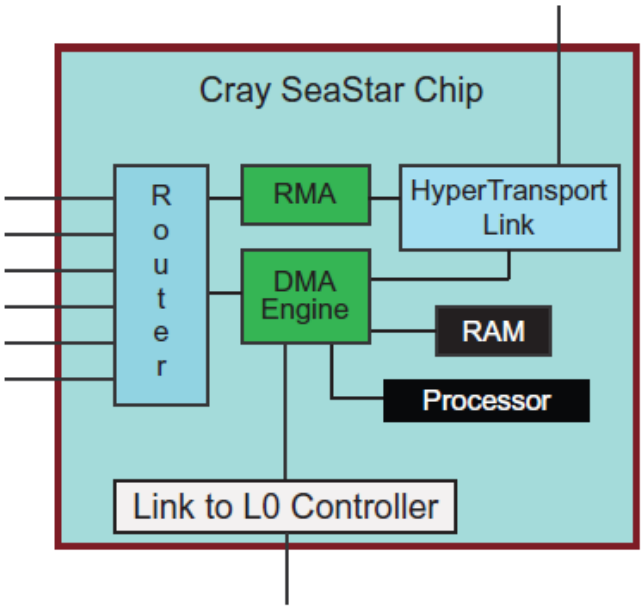


Figure 2-3 Cray Seastar chip
Image Source: The Cray Supercomputing company (1 p. 19)

As shown in **Figure 2-3**, Each Seastar chip consists of a HyperTransport link which connects Seastar to AMD Opteron Processor. Direct Memory Access (DMA) Engine, does all the movement of memory from and to the node memory. There is a 6 channel Router for connecting to the other Seastar chips on the network. A Link (L0 controller) to the Blade control processors is present, which is used for booting maintain and monitoring the systems. This Interconnect uses a low level message passing API called Portals, which is partly installed on the Seastar firmware. There is an embedded PPC 440 Processor and an on board 348KB RAM. The firmware runs on this embedded processor Seastar. For more details about Seastar Interconnect refer (3).

2.1.3 Gemini Interconnect

The new Cray XE series systems will be carrying Gemini Interconnect instead of Seastar chips on their nodes. This new Interconnect offers lots of upgrades, but still the main important feature of the Gemini ASIC is the support for Global Addressing Space and advanced NIC design to efficiently support one-sided communication. This provides a great scope of improvement for all one-side communication based libraries.

Each Gemini ASIC has two NIC's and a 48 port 'Yarc' Router. These two NIC's are in turn connected to two different nodes using separate Hyper Transport 3 Interface (HT3), therefore each Gemini chip acts as two nodes on the 3D Torus network. As shown in Figure 2-4, the NIC's are connected to router using a Netlink block. This handles the clock differences between NIC and Router. As in the Seastar interconnect, a supervisor link is used to connect to the L0 Blade Control Processor.

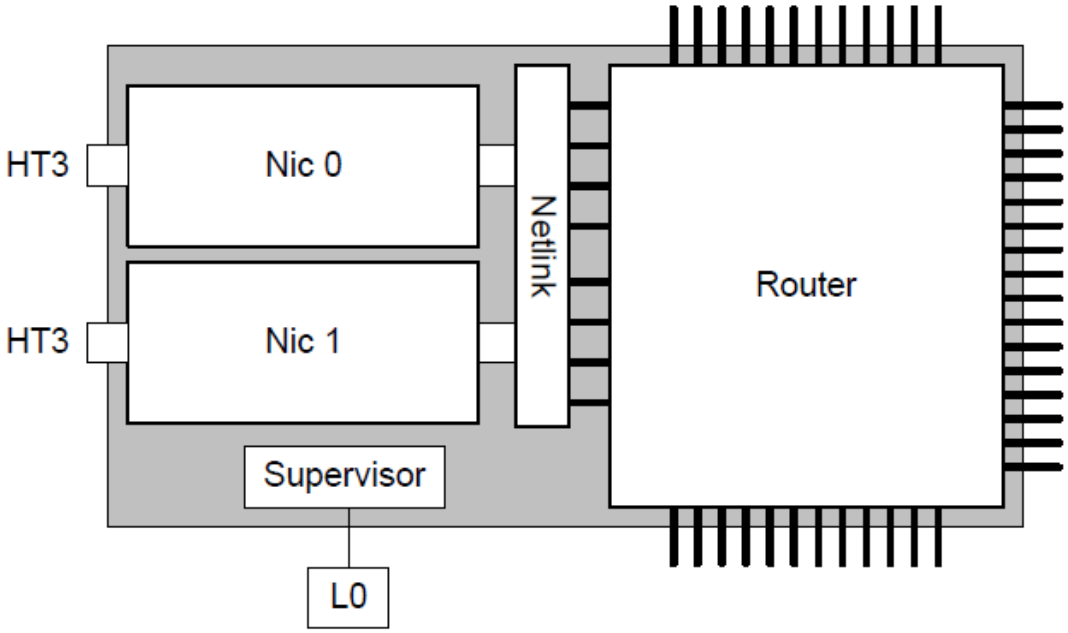


Figure 2-4 Block diagram of Cray Gemini Interconnect
Image Source: The Cray Supercomputing company (4)

Gemini NIC

Each Gemini NIC provides the following mechanisms (4).

- Fast Memory Access (FMA) is a mechanism where the NIC translates the data stored by processor on NIC to a fully qualified network level operation. This provides low latency for small messages.
- Block Transfer Engine (BTE) which supports asynchronous block transfers between local and remote memory in both directions. This is used to transfer large blocks of data.

Gemini Performance

Speed of each Gemini NIC is 650MHz; the router operates at 800 MHz. Each Gemini Chip with 48 ports has 160 GB/s switching capacity. It has 10 12x Channels to connect to the network as compared to the 6 channels in Seastar chip (4).

2.2 Software Layers

2.2.1 Global Array Toolkit

Global Array Toolkit is a library developed by Sandia National Laboratory which provides an efficient portable shared memory programming interface for distributed memory systems. At any given point each processor has a global view of the data (as

shown in **Figure 2-5)** and any part of the GA data can be accessed independently by any processor. It is also compatible with MPI and many other 3rd party software packages.

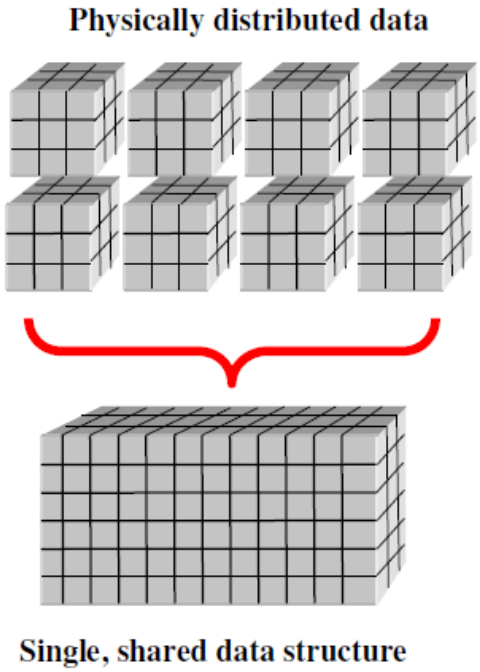


Figure 2-5 View of GA Data Structures
Image Source: Global Array's, a shared memory programming toolkit (5 p. 4)

The shared memory model on Global Array's combines the advantages of distributed computing model with ease of use of shared memory programming. GA exposes the fact that remote data is slower to access than the local data. The data locality can be found out by the use of function call which helps the user to exploit the SMP locality to achieve peak performance. GA also allows the user to control the data distribution, which may not be necessarily used though. To ensure memory consistency GA requires a call to Memory Fence or Barrier Synchronization function, if the globally visible data is modified. Blocking operations that do not overlap on same target or access same arrays can complete in arbitrary order. Also the non blocking load and store operations can complete in any order, though ordering of these operations is possible by using wait or test calls.

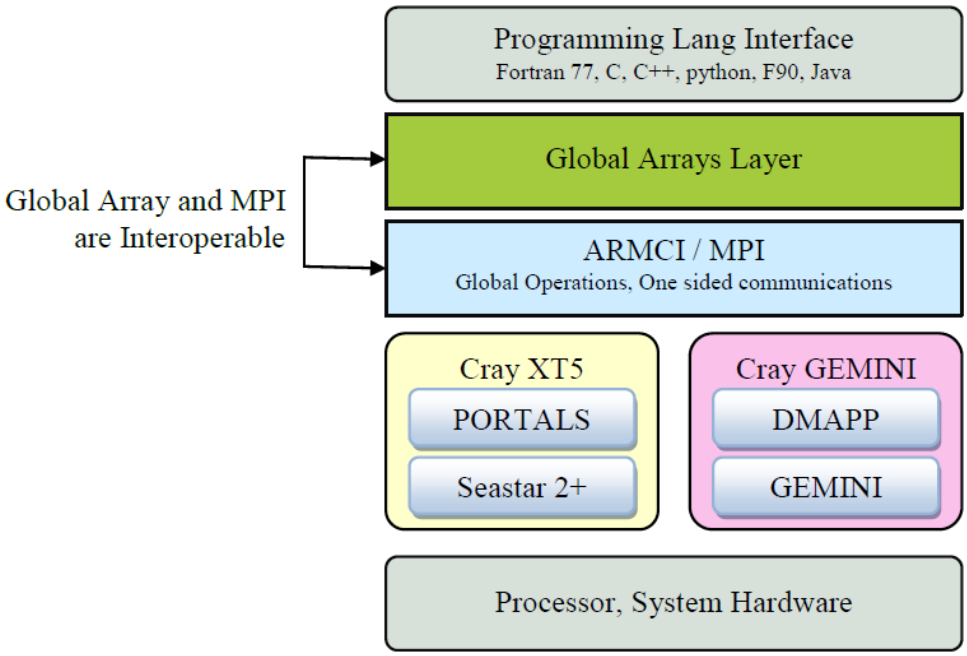


Figure 2-6 Overall structure of GA Toolkit with different Language bindings

GA uses ARMCI as its primary communication layer, but the elements of job execution environment like job control, process creation are managed by message-passing library. Since GA is entirely dependent on ARMCI, Performance of GA is proportional with ARMCI Performance. Porting GA to any new platform is simply associated with porting ARMCI to that platform. Each GA function call is eventually translated into multiple ARMCI calls(6).

Apart from the normal features data movement and atomic operations, GA Toolkit has many advanced features. Some of them are support for ghost cells, global transpose, sparse data management, Support for global I/O and Mirroring (shared memory cache for distributed data). Because of these advanced features GA is widely used in several large applications like Quantum chemistry code (NWChem, GAMESS), Molecular dynamics, Computer graphics (parallel polygon rendering), Computational fluid dynamics (Ludwig, NWPhys/Grid), Atmospheric chemistry, etc (7).

2.2.2 ARMCI

Aggregate Remote Memory Copy Interface (ARMCI) is a general, portable and an efficient one-sided communication interface (8). It provides extensive RMA operations like Data transfer operations, atomic operations, synchronization, Memory management and locks. ARMCI offers transfer of non-contiguous data transfers in both blocking and non blocking versions. Though ARMCI can be implemented on many type of

architecture, this report explains only the implementation Cray XT systems with Seastar and Gemini Interconnects.

In General, the ARMCI has to do the following 3 fundamental tasks (9).

- Prepare the memory for communication; this involves allotting all remote memories used for ARMCI Communication to be allotted using the ARMCI_Malloc library call. This enables the ARMCI library to prepare for remote communication to and from that memory.
- Use Portals/uGNI[7] communication calls to implement the data transfer operations. If the data transfer operation is a Vector or Strided one, it is then translated into a series of Network level Get or Put calls.
- Implement atomic read modify write operations

Since Global Array Toolkit's communication entirely depends on ARMCI, optimizing ARMCI becomes one of the important tasks of this project. More about ARMCI's implementation and working is explained throughout the report.

2.2.3 Portals

Portals is the Network Programming Interface developed by Sandia Laboratories and University of Mexico for communication between nodes of parallel Computer (10). This is lowest level Network Interface available in Seastar Interconnect of the Cray XT Systems. This provides one sided communication API for data transfer among the processor.

Portal provides three major data movement operations Put, Get and GetPut. Due to the lack of direct global addressing or one sided communication support on the hardware it is highly not possible to do a data transfer operation without the involvement of the remote processor. So portals get call, which fetches the data from the remote processor by itself, is a two sided operation. This requests the Interconnect on the remote node for the required data. And the remote interconnect either fetches it directly from memory or fetches it with the help of the remote processor and sends the data back to the client.

Portals addressing is effectively done with the set of special values called Match bits. These bits hold the exact offset or location of data on the remote processor. So with the portals data transfer call these bits are sent to the remote processor, which with the help of these bits identifies the exact data needed and transfers it back to the client. Thus providing a simple data transfer mechanism. Apart from this portals also provide various other options like blocked polling (Section 4.2.2), attaching actual data's

[7] uGNI – User level Generic Network Interface

address to the memory descriptor thereby ensuring easy data transfer into the client memory. Portals also bypass's OS for almost all the data transfer operations. For more details about the implementation and working please refer (10).

2.2.4 DMAPP or uGNI

uGNI along with DMAPP forms the lowest level Network Programming Interface on the new Gemini Interconnects(11). ARMCI's is implemented on the Gemini Interconnect over the layered interface of both uGNI and DMAPP, as shown in Figure 2-7.

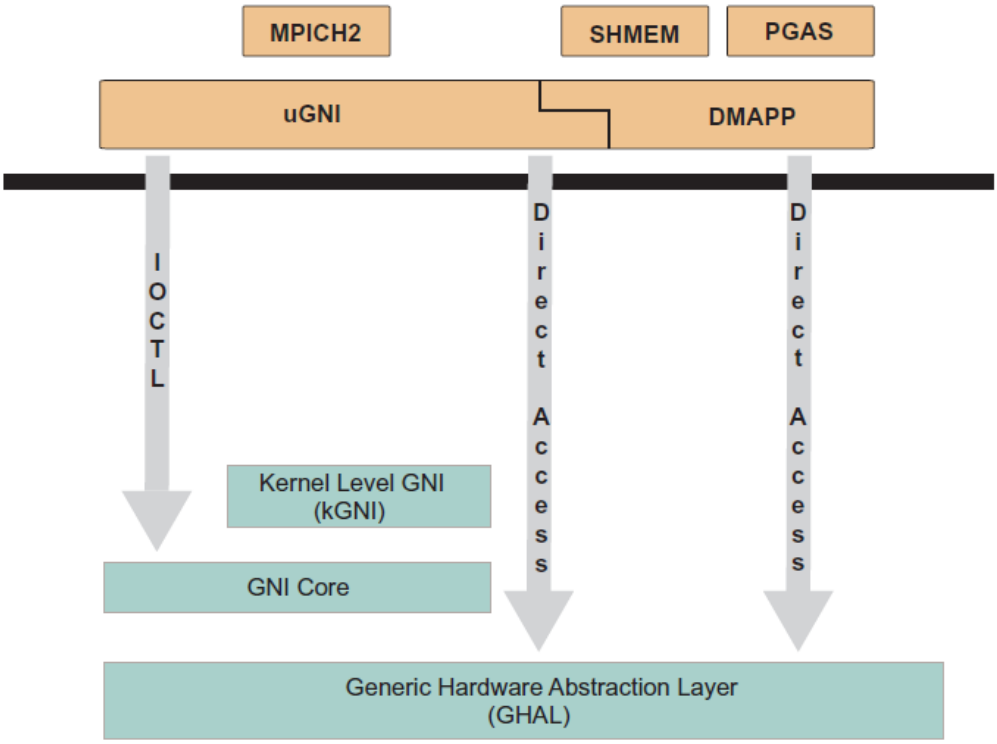


Figure 2-7 Software Stack of GNI and DMAPP API Layers
Image Source: The Cray Supercomputing Company (11 p. 23)

The availability of FMA[8] helps the uGNI to pipeline many small messages to extract a very low latency time. uGNI uses Remote Direct Memory Access(RDMA) to create communication between end points. It also helps in registering memory for use by the Gemini ASIC.

[8] FMA – Fast Memory Access

DMAPP includes blocking and non blocking data transfer one sided operations. This makes effective use of the available hardware support for Global Addressing. Apart from the normal put and get operations DMAPP also supports indexed variants of the put and get operations, this helps in direct accessing of the remote memory rather than trying to fetch the data by sending a message and in turn receiving it back from the remote processor.

DMAPP also supports Scatter and Gather Operations. It can be used by all the applications to build their own set of message passing mechanism over it. The FMA of the Gemini Interconnect has a high packet rate; there by flooding the network with more massages will not have any adverse traffic effects on the network. For more details on uGNI and DMAPP refer (11).

2.3 Previous Work

Gemini Interconnect is very new to the market, so there are no previous works as such done on this Interconnect yet. But there a quite a few works had done on the ARMCI.

(12) Shows the performance gain obtained by ARMCI various networks. The results indicate that performance of ARMCI is as close to the MPI. So this is an encouraging part for the project.

(13) Show the works done on ARMCI to enable it scale to high number of cores on very large massively parallel systems.

Chapter 3

Initial Benchmarking

Initial benchmarking is done with 2D Matrix transpose program and it's used for timing the basic data transfer operations of the toolkit. Matrix transpose is preferred because it does not have any calculations to do, so that it gives the real estimate of the communication part of the toolkit. It is also helpful in understanding about the Interconnects performance, which is the main area of concern in the project.

In the later stages of the project, a DFT module from NWChem package is used to estimate and understand the performance on a real time code.

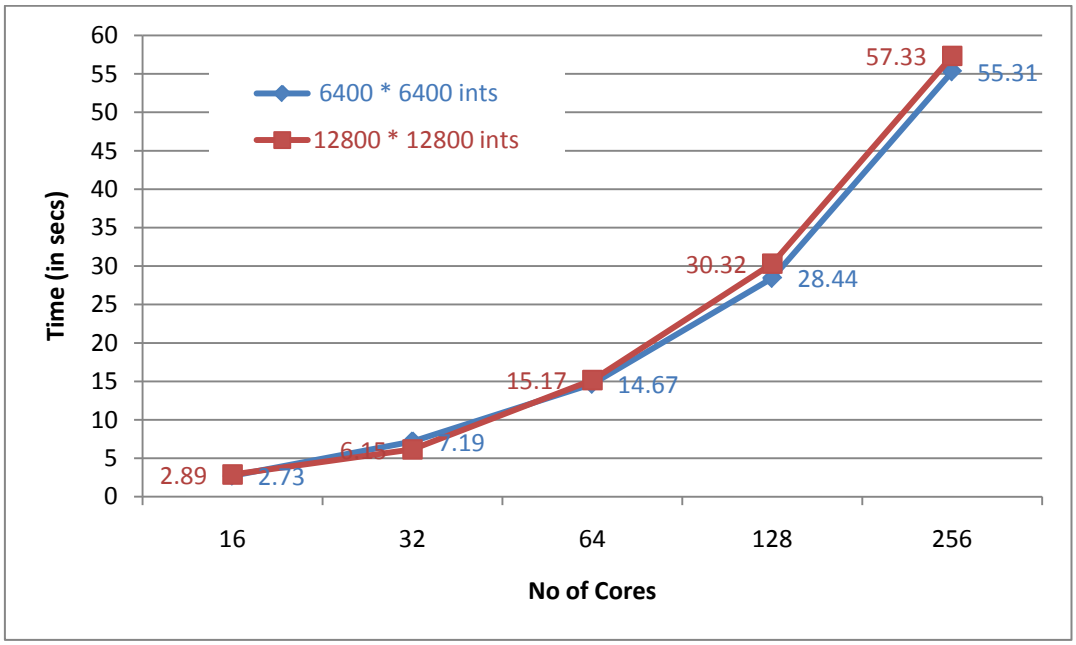
3.1 2D Matrix Transpose

A 2D Matrix transpose exercises communication with pairs of processors where each processor communicates with all the other processors simultaneously. In Global Array Toolkit the array is normally split equally across the entire range of processors. So a transpose operation involves communication with all processors in the group. This gives an estimate of the total communication handled by the toolkit as well as the interconnect. The size of the problem is decided by varying the size of the matrix.

Global Array Toolkit has an inbuilt matrix transpose function *GA_Transpose* which is a collective operation. The implementation of this library function is well optimized. But the implementation of this function is opposite to that of the naive version. In both the versions the array is split either row wise or column wise across the processors. During transpose, blocks of data is read from the global array using GA_Get, this might be local operation or global operation based on the way the array is split. After transposing this data locally, it is written back to the global array; this operation again depends on the way the global array is split.

In naive version, the array is split across rows. A column wise remote get operation is performed using GA_Get call. And a row wise local put operation for storing the transposed matrix is done using a local call to GA_Put. In GA_Transpose library the calls are reversed. GA_Get call is local and GA_Put function performs a remote store

to the global array. So this is useful for comparing the efficiency two basic one sided operations Put and Get.

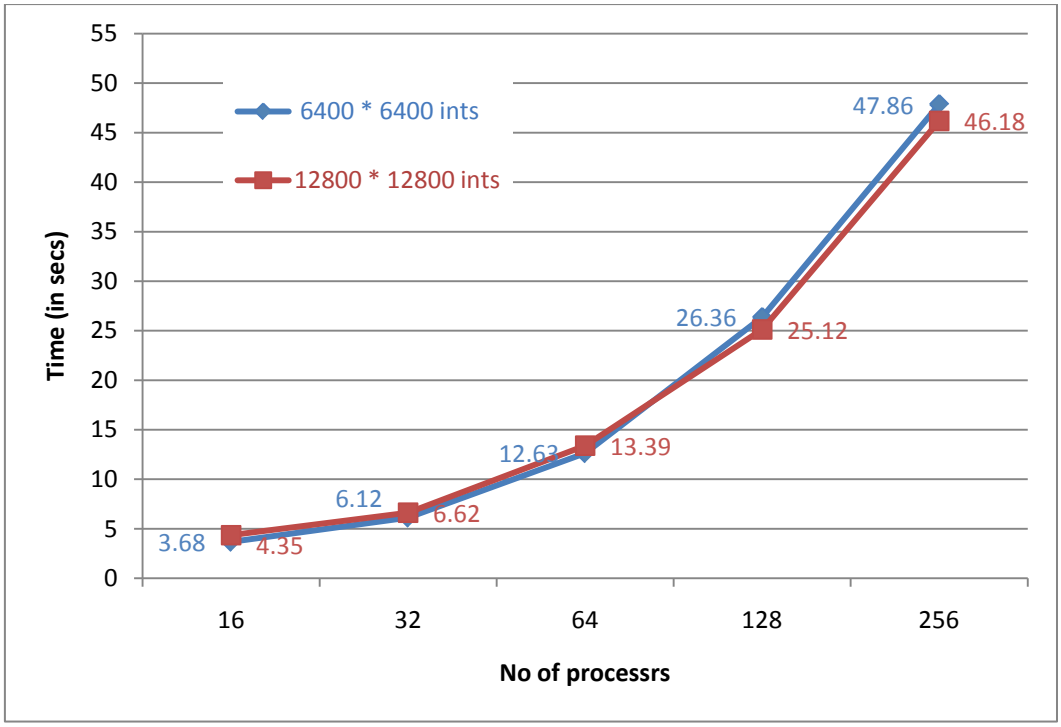


Graph 3-1Matrix transpose (Naive Implementation)

Graph 3-1 shows the plot for Time vs processors for a grid size of 6400x6400 ints (156MB) and 12800x12800 (625MB) of data. The time delay as the processor increases is due to the communication cost. When the number of processor increases the size of each message decreases but the number of messages sent increases. For 256 processors the message sent by each processor reduces to less than 10KB. The bandwidth of the seastar chip is around 57.6 Gb/s and the insertion bandwidth of the hypertransport is 6.4 GB/s, so this communication time is just the intiation cost or latency.

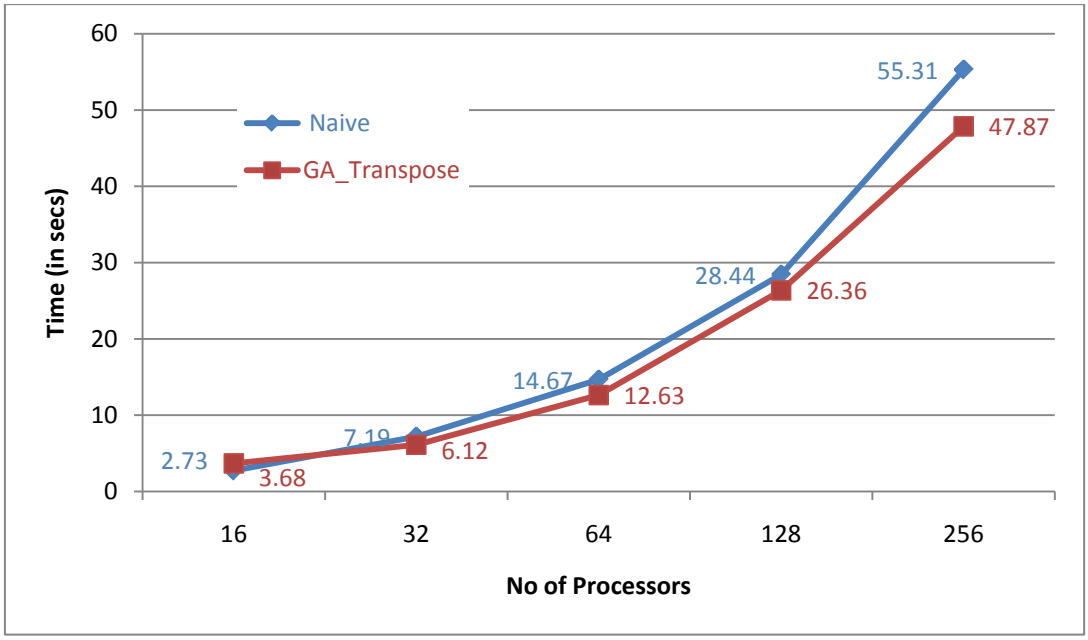
As said in Chapter 2.2.1, GA is built on top of ARMCI, which is a one-sided library. Since Seastar chip does not support one sided communication on hardware, there is a server running on remote processor which serves as a receiver. But to the programmer it appears as a one-sided message passing. This server is programmed in such a way that it blocks during network events instead of polling so that it does not waste any processor time to poll while waiting. The message is packed in the compute kernel of the client before it is sent through portals and the data server unpacks the message after recieving and writes it to the physical address of the remote processor. So this packing and unpacking is an additional hidden cost for all messages along with the normal latency of the network. Since this has to pass through two software layers from GA to ARMCI and then to Portals, this is expected to be slightly higher than normal MPI.

In the library function GA_Transpose, the performance is almost same when the message size increases. As the processor increases the communication cost also increases which affects the overall performance, as shown in Graph 3-2. For both sizes of grid the performance is almost same proving that communication cost is more than calculation cost.



Graph 3-2 Matrix transpose using GA_Transpose

Even though for both the versions the communication cost is higher than the calculation cost, there is a significant difference with the timing for 256 processors. The performance is same for small number of processors but when the number of processor increases the performance gets better for the library function than the naive version. This difference is mainly because of the put and get variation in their implementations. In the library function remote put is used and in naive version remote get is used. For smaller messages put turns out to be much better than get which is evident from the Graph 3-3. There is a huge difference of around 8 secs for 256 processors which is 10% higher than naive implementation. This gives a real chance to optimize the get functions implementation to match the performance of put, which is explained in Section 5.2.



Graph 3-3 Naive (vs) GA_Transpose version of Matrix transpose for a grid size of 6400 x 6400 integers

All the further optimizations done in the project are analysed and compared with the above set of benchmarks.

3.2 NWChem

NWChem is the most widely used computational chemistry package, which is built using Global Array Toolkit (14). Benchmarking using this application will give the estimate based on the real time use of the application. In real time applications there are more incurrence of Get and Put functions in the random fashion. This is helpful in analyzing the memory and buffer usage of the toolkit.

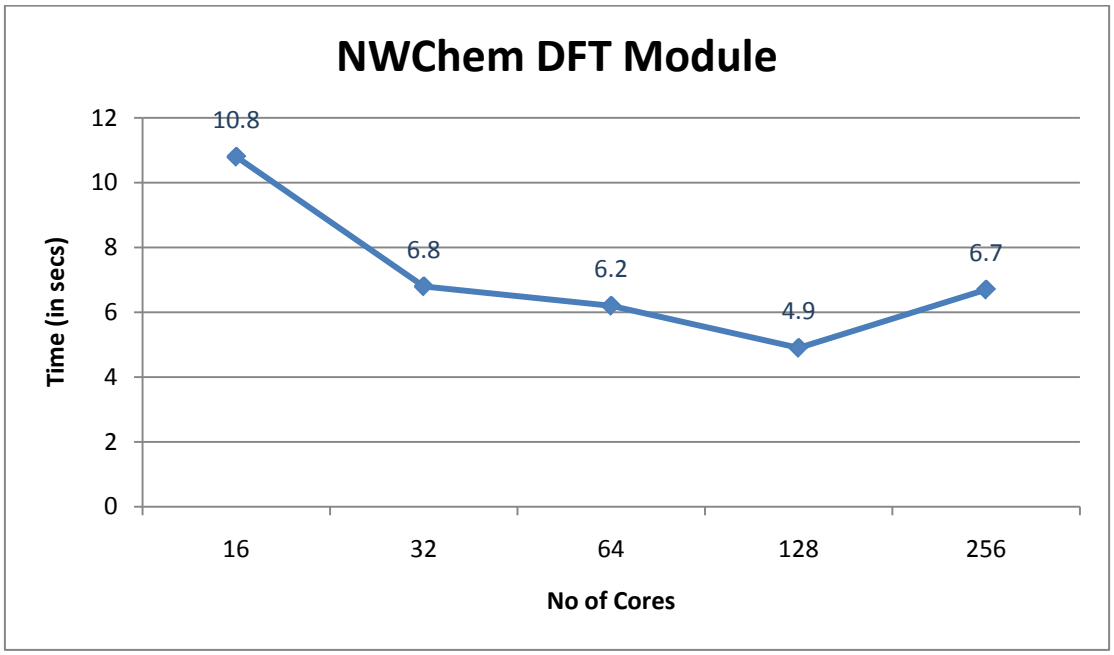
A module M05-2x from the Density Function (DFT) of the form Hybrid meta-GGA for the equation NH2 + CH3 is selected for benchmarking purpose (15). The project is not concerned with the application of this module. The area of interest is the use of GA calls in this benchmark. The GA statistics for this module is shown in the Table 3-1.

The usage of 256 global arrays is indeed a good test for the memory and buffer size of the toolkit. This aspect of the toolkit has a major concern on the performance during the communication. Since there is no active involvement of the remote processor during one-sided communication, the receiving would be quicker provided there is enough memory already available in the toolkit. Rather than waiting to free up the memory for the arriving message.

	Create	Destroy	Get	Put	Accumulate	Scatter	Gather
Calls :	256	256	3425	465	448	0	0
Total Memory transferred in KB :			7031.25	1796.87	345.70	0	0
Remote Memory transferred in KB:			5332.03	1191.40	236.32	0	0

Table 3-1Showing the GA Statistics of the selected NWChem module for 256 cores on processor 0

Number of Get functions used in this Module is much higher than the number of Put. Since the get calls are facilitated by the client processor, there will not be any memory synchronization problem. There is chance for shortage in the number of request buffers since there is high number of get calls used. Number of buffers allotted in contrast to the size of each buffers has less impact on the one-sided get and put calls but it will have more impact on the Scatter and Gather calls, as there are more calls initiated on the same processor simultaneously.



Graph 3-4 Showing NWChem dft_m05nh2ch3 module for 4 core node

This NWChem module is run on many range of processors with 4 cores each on node. This NWChem module is used only for benchmarking purposes. For all the other tests

and experiments, the Matrix transpose program is used. As shown in Graph 3-1, as the number of cores increases from 16 to 32 there is super linear speed up with time dropping down rapidly. This is attributed due to the calculation part being equally divided among the cores. As the number of core increases to 64 there is not a major time difference. Even if there is a difference it is due to the extra buffers available. For 128 cores there is again a time drop, due to the reduction in the size of each message. Message division (among the processor) reduces the size of each message enabling it to fit in small buffer size. This attributes the message to transfer in eager protocol. Eager Protocol for small message size is usually faster than Rendezvous protocol for larger messages (Section 4.1).

After a certain point communication time takes over the calculation time, this happens at 256 cores. Here communication becomes more prominent than calculation resulting in increase of time in comparison with 128 processors. On the contrary the total number of messages sent by each processor in 256 cores will be decrease slightly from 128 cores due to the fact that total size of data transferred has now been divided among more number of cores. This NWChem module has many important features necessary for a good benchmarking code.

3.3 Conclusion

Initial Benchmarking needed for estimating the performance of the toolkit has been done. Matrix transpose program will be used as micro benchmark for testing and experimenting purposes. The values obtained from this chapter will provide as a guideline for comparison in the rest of the chapters to follow.

On the other hand NWChem benchmark is used as a benchmark for testing in real time applications. This gives us a different measure of the code, like it buffer and memory usage. It also gives the estimate about the conjunction in the interconnect, by sending more messages than the former matrix transpose benchmark.

Chapter 4

Optimization Strategies

4.1 Eager vs. Rendezvous

There are two major protocols when it comes to message passing, Eager and Rendezvous. The efficient use of these protocols might have severe effect on the performance of the Global Array Toolkit.

EAGER PROTOCOL

In this protocol the client sends a message to the remote processor and finishes the call on client side even if there is no matching receive call in the remote processor. In ARMCI the remote data server is responsible to process the incoming messages. If the message size is small then the remote data server stores the incoming message to a local temporary buffer if the actual physical memory where the data has to be stored is not ready. Once the physical memory is free, then this data is copied back from the temporary buffer to the actual address.

This type of protocol has its own advantages and disadvantages. If the message size is very small then the synchronization cost at the remote server will be more than actual the data transfer operation from the temporary buffer to the actual address. In that case this protocol will reduce those synchronization delays.

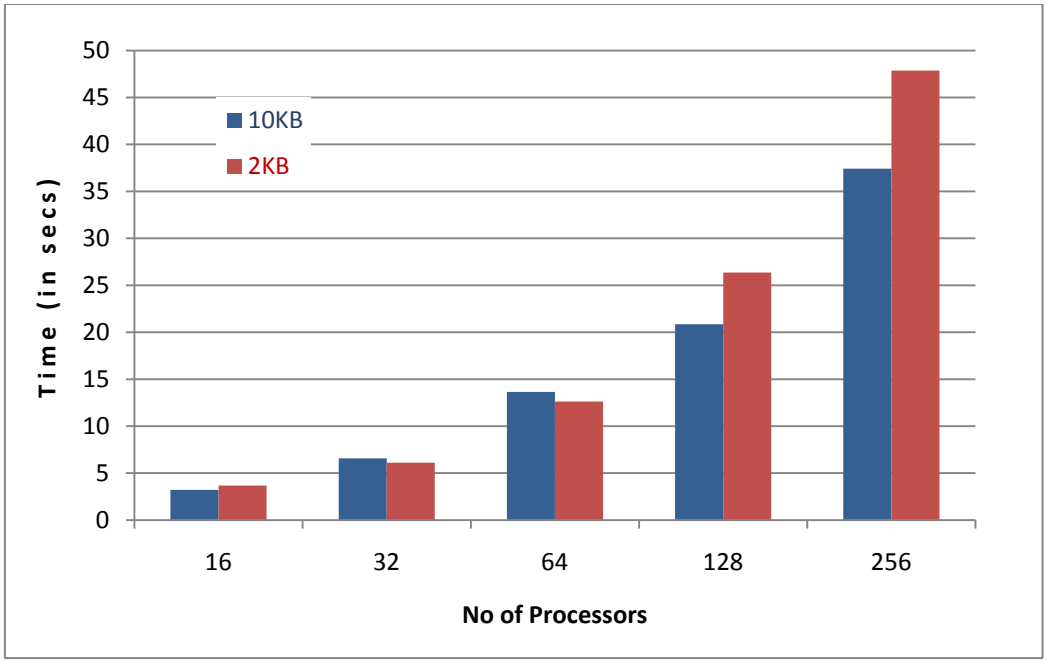
RENDEZVOUS PROTOCOL

In this protocol only the message header is sent to the remote processor. And when the remote processor is ready to receive the data, the whole data is sent. This protocol is useful when the message size is too large to be buffered and copied. Only the message header needs to be buffered, thereby saving lot of time on data copy for large messages. But this method increases the synchronization and the number of messages sent, but this time is negotiable when considering size of the data transferred.

ARMCI's technical implementation of the rendezvous protocol is slightly different. But the logic and working is almost the same. Deciding between the Eager and Rendezvous protocol is critical when considering the communication time on massively parallel machines like HECToR, where there are more inter nodal communications.

4.1.1 GA_Transpose Implementation

In our Matrix Transpose Benchmark on 128 processors with 4 cores on each node, each processor has to transfer around 10KB of data to each other processor. In GA_Transpose implementation the remote call is GA_Put, so 10KB of data has to be sent to from client to remote processor. The protocol used in this communication is very critical to the performance. The entire initial benchmarking is done with Eager Cut off limit of 2KB per message. On 256 processors the message sent by each processor is about 2.5KB which still does not fit in the Eager buffer size.

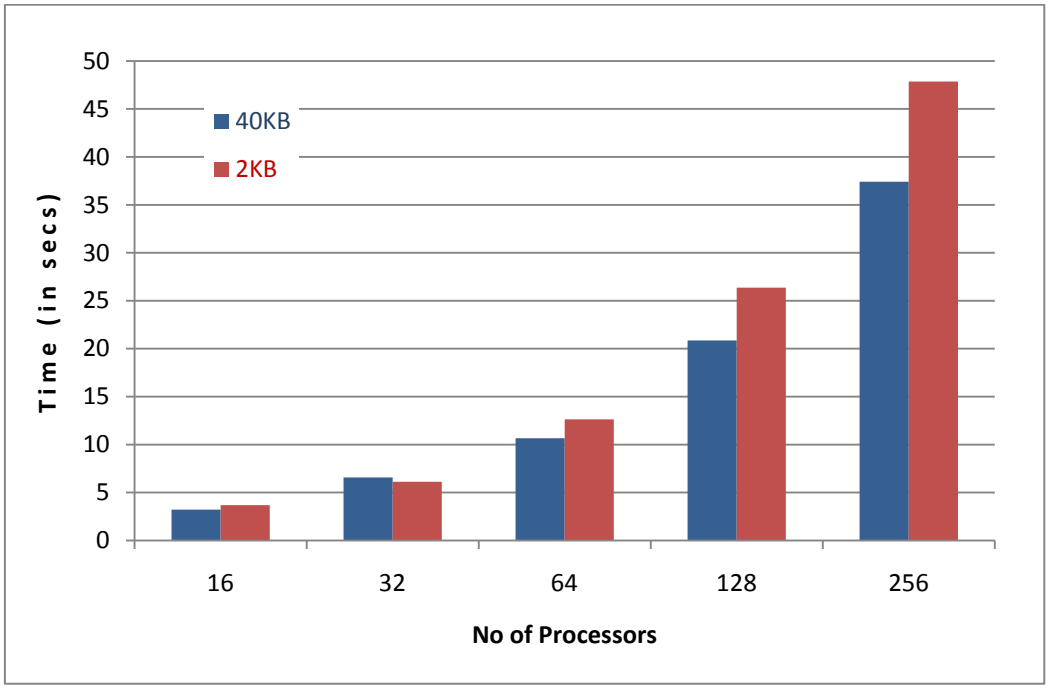


Graph 4-1Showing GA_Transpose (6400 x 6400 ints) implementation using remote Put operation with Eager limit of 2KB (vs) 10KB

In the Rendezvous protocol for ARMCI_Put call (Which is responsible for GA_Put), the message header for the data which has to be transferred is first sent to the remote processor. On receiving this message the data server on the remote processor issues a Network level Get call to fetch the data from the sender. During this process of fetching the data, the remote processor is idle waiting for the data to be received. This time delay in the rendezvous protocol is avoided in the Eager implementation where the data along with the message header is sent as a single message. There is no fetching call from the data server on the remote processor, thereby saving a lot of time during communication.

On increasing the Eager buffer size to 10KB, the data transfer protocol changes from Rendezvous to Eager for both 128 processors and 256 processors. Blue lines on the Graph 4-1indicates the time for GA_Transpose implementation of Matrix Transpose with Eager limit of 10 KB. As the message gets fit into the Eager limit there is only one message sent and no synchronization costs as well, this is reflected directly on the performance. As can be seen for 128 and 256 processors the time taken has reduced considerably around 10 seconds for 256 processors and around 5 seconds for 128 processors(4).

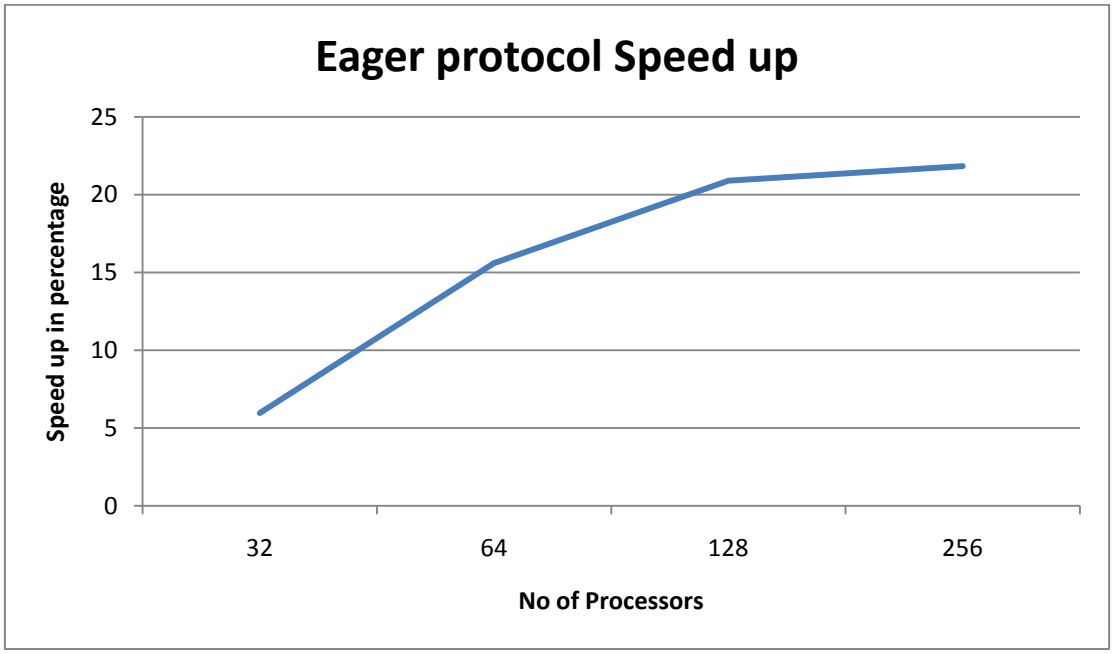
In Eager Protocol, when the Eager message is sent and even if there is no matching receive on the data server, then the data is buffered and then copied later. Cost of this data copying from temporary buffer to actual address of around 10 KB is still lesser than the synchronization cost accomplished during the rendezvous protocol.



Graph 4-2 Showing GA_Transpose (6400 x 6400 ints) implementation using remote Put operation with Eager limit of 2KB (vs) 40KB

But if this limit is further increased to match the 64 processor's message size of 40 KB, then the cost of copying 40KB from buffer will be more than the previous cases. This may result in no performance gain. In this case of matrix transpose the processor does not have any calculation to do, so the processor is idle for most of the time. This can help in these data transfer from the buffer but if the processor has some other calculation to do this data transfer operation will be consuming more useful processor time. So it is not a sensible way to increase the Eager limit to 40 KB. And also as seen from the Graph 4-2, the performance gain in time is around 2 seconds, which is very less when compared to the previous cases.

On extending this Eager limit to suit the message size of 160 KB for 32 processors, there is almost no speed up in time. As expected the time for copying buffer data is more than the synchronization time when it comes to less number of processors. As seen in Graph 4-3 the speed up is rapidly taking a deep as the processor count decreases, this is because as the processor count decreases the buffer size increases.



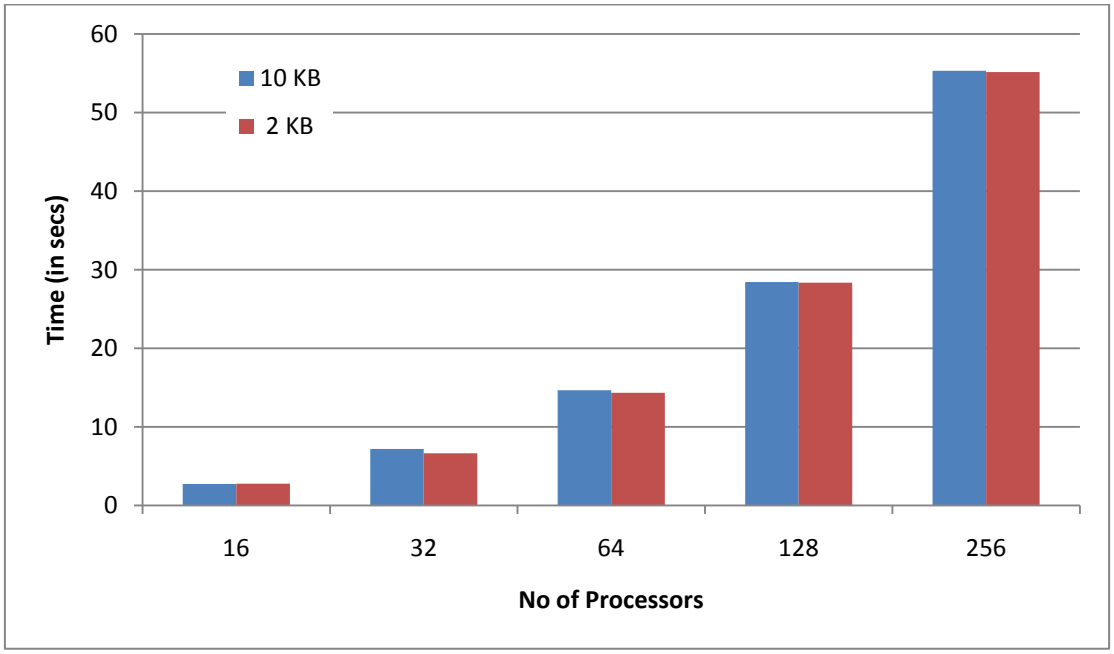
Graph 4-3 Showing speed up in percentage when the message is transferred in Eager Protocol

4.1.2 Naive Implementation

In the Naive implementation of the matrix transpose, the remote transfer operation is performed using GA_Get call. The amount of data transferred is same as the GA_Transpose since it is the same grid size. When the Eager Cut off limit is increased from 2KB to 10 KB, there is an expectation for certain change in the performance. But as seen from Graph 4-4 there is no major change in the timing and performance. In the graph the blue lines show the timing for modified Eager limit of 10 KB and the red lines show the performance for default Eager cut off limit of 2 KB.

This shows that for both small and large messages GA_Get will consume the same amount of time as long as the whole message fits into the range of the 2 GB which is the limit of a single message sent using portals. So for smaller messages GA_Get is surely not the choice. This is due to the underneath implementation of ARMCI_Get, which is responsible for GA_Get. Since there is no one-sided communication supported on the Seastar hardware, the Get operation is performed as a two sided operation which

involves both the client and remote processor to serve the request. So there is no proper implementation of Eager Protocol for ARMCI_Get. So there will be no change with respect to the Eager cut off limit in GA_Get function.



Graph 4-4 Showing Naive Implementation (6400 x 6400 ints) for Eager limit of 2KB vs 10 KB

So this result shows that there is not much difference with the ARMCI current implementation of Eager and Rendezvous. More details about ARMCI_Get implementation is given in the Section 5.2.

4.2 Communication Helper Thread and Buffers

4.2.1 Request Buffers

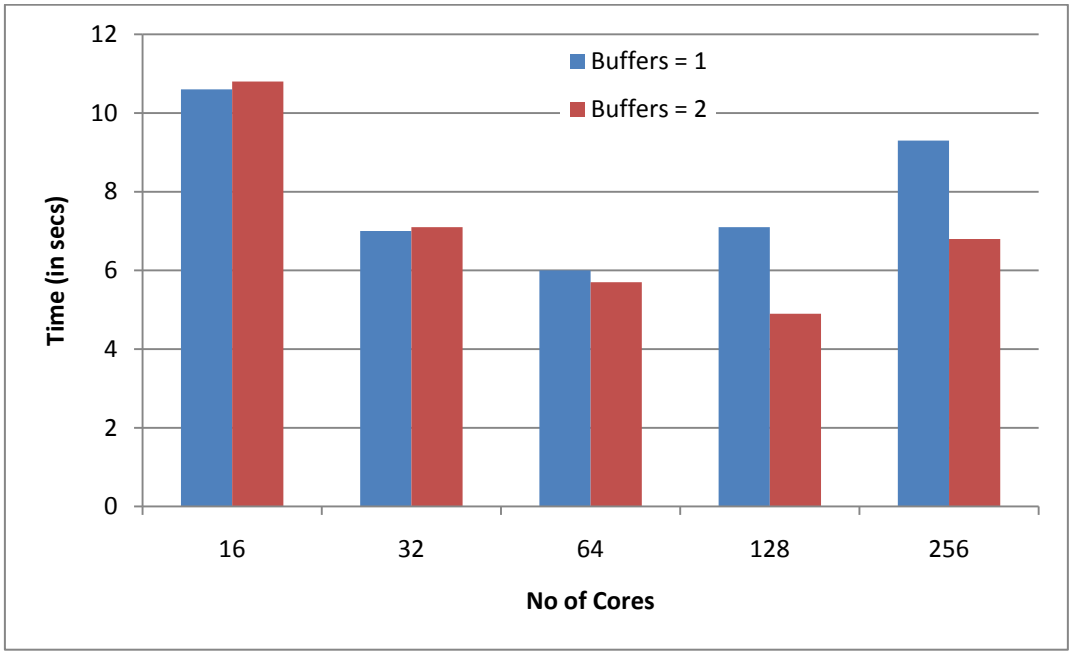
There are two different buffers on the ARMCI implementation of the message passing. CHT buffers (Buffers in Data server) on remote processor, which are used for receiving messages and data. Client buffers are used by the client processor during communication with the remote processor or remote data servers.

4.2.1.1 Communication Helper Thread (CHT) Buffers

The Communication Helper Thread acts as the data server in the remote processor, which is responsible for receiving messages (explained in Section 4.2.2). This server has to have at least one unacknowledged buffer for each client processors. So it should

have a minimum of n buffers where n being the number of processors. Deciding on this number of buffers is very critical for ARMCI Performance. For memory intense applications, the program needs more memory for computation therefore buffer memory has to be minimum for those applications.

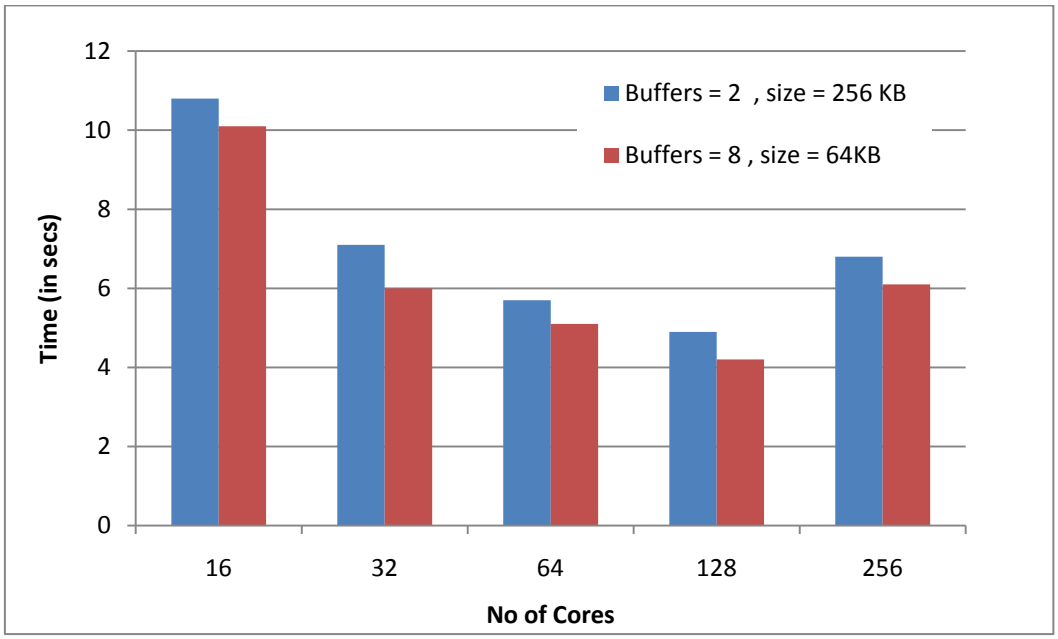
The DFT module of NWChem is run with the minimum of 1 buffer for each client processors. As expected the performance is far lesser than that for the 2 buffer for each client processor. When the buffer is limited to just one, then it cannot process anymore request from that client until it completes this transfer. Only then it sends the acknowledgement which makes the client processor to send the next message in the queue. This helps in the scalability of the application. As seen from the Graph 4-5 for less number of processors there is almost no change in time with respect to the number of buffers.



Graph 4-5Showing NWChem DFT module's timing for CHT buffer size of 1 (vs) 2

Mainly because of the amount of calculation which is higher than the communication, there is no effect of communication wise optimizations. This range continues till 64 cores, but on 128 cores there is a significant increase in the performance for 2 buffers. This is where the communication part is higher than the calculation part and hence the optimizations done for communication results in better performance gain. For 128 cores there are more messages sent by each processor to other, since the total data gets divided among many processors. In this case increasing the number of buffers results in smooth and continuous flow of data, without halting on the remote server for synchronization. This is same for 256 processors as well. But in 256 processors there is still no speed up when compared to 128 processors. As said in Section 3.2 this is because the communication part is higher than the calculation part.

Each buffer has a size of 256KB. So for a single node there are 4 clients and each client has to have 2 buffers each for 256 processors, so totally 512 CHT buffers. For a single node (512*4) buffers need to declared, each of size 256KB. So the total minimum CHT buffer memory needed on a single node is (512*4)*256 = 512 MB. This size is very less for a buffer size on a shared memory of about 8GB Per node. So for this module this buffer size can still be increased further.



Graph 4-6Showing NWChem DFT Module timing for CHT buffer size of 2 (vs) 8

The Values obtained on increasing the buffer size to 8 (64KB each) from 2 is shown in Graph 4-6. As can been seen there is again a considerable amount of speed up obtained from increasing the number of buffers. The memory fence operation is implemented using the combination of CHT acknowledgements and source credit obtained from direct data transfers for contiguous messages. For non contiguous data transfer using CHT, the performance is entirely dependent on the Number of buffers and its size.

For 8 buffers of 256KB, the total buffer size needed on a node for 256 processors is about 2GB. So this is about 25% of the total memory size which is way too higher and when the number of processor still increases this buffer size will still climb higher. For matching with the high number of buffers, we reduce the buffer size to 64 KB. This gives a total buffer size of 512 MB, which is about 5% of the total memory available on the node. Increasing this memory further will affect the memory intensive applications performance. So deciding on the CHT Buffer size is a critical factor and decisive factor on the communication part of the toolkit.

4.2.1.2 Client Buffers

The Client buffers don't have much impact on the performance, but it is useful in initiation of the communication. Each client processor has its own set of buffers allotted, which it uses for communicating to the Data server on the remote processor. In one sided communication there is no need for any buffer memory on the processor which initiates the call, since the actual physical memory to which the data has to be written is already free and available. But in this case, the Data server on the Remote processor sends back an acknowledgement for all the messages which it receives.

This acknowledgement is better way to keep track of the messages sent, which allows the client processor to know about the status of the message sent to a particular node. It helps in deciding whether or not to send further message to that node. Also this buffer is used during the put operations whereby the data is copied into the client buffer before sending, so that the actual address can be used for other purposes and need not wait till the operation completes. The maximum number of client is fixed to the total number of processors, so it can invariably extend till it needs.

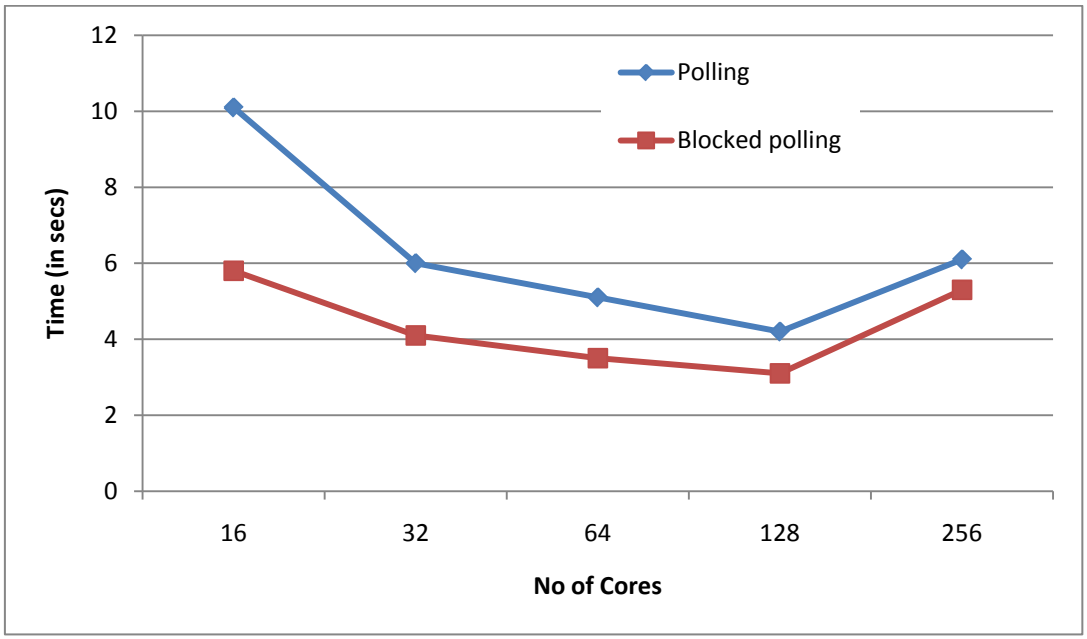
4.2.2 Communication Helper Thread (CHT)

Communication Helper Thread (CHT) or otherwise called Data Server is the part of the ARMCI which helps in the completion of the remote data transfer operations. This Server runs on the background on any of the single core in the node. If there is more number of cores available in a node than used, this data server is run on the core free of calculation. But in the normal case where all the cores in the node are used by the program, this data server is allotted to the lowest ranked core on the node. This server is spawned separately using the system level call clone, thereby acquiring its own portal network communications and a separate clean stack. This is the same case with the Gemini Interconnect as well.

The CHT is responsible for receiving data's sent for all the processors on the node. This receives the data and formats them based on the specified options like contiguous, Strided, lock. CHT also has the capability to block during the network operations by interrupting without polling. If the CHT is polling for the network then it will be wasting valuable CPU time by just waiting during the network operations. Therefore blocking this communication thread when there is no events in the network receive queue saves lot of CPU time. In Cray version of Portals this can be the environmental variable CRAY_PORTALS_USE_BLOCKING_POLL is set to achieve this blocking poll operation. This has a considerable effect on the performance.

Without blocking the thread, the processor will not be able to send any messages when they are receiving from other nodes. With CHT thread the processor can send a certain limit of messages asynchronously at any given time. This can be accomplished even

when there are many messages in the queue for receiving. There by giving each processor a fair chance in the message passing. Since the CHT has its own portals initiated this method of asynchronous sending and receiving is possible. Number of messages (n) sent asynchronously by a processor at a particular time is fixed. And the processor can send n+1th message only after receiving an acknowledgment for any of the previous n messages, thereby not flooding the network. This also maintains fair communication synchronization among all the cores in the node.



Graph 4-7Showing NWChem DFT module for Blocked (vs) Normal Polling

As can been seen from Graph 4-7 the blocked polling option in the CHT has a significant impact on the performance for all the processor. This surely proves that without blocking the Communication thread is utilising the essential computational time of the processors. On blocking this thread when there is no message to receive in the queue leaves the CPU to do the computational part of the problem, thereby effectively using the valuable CPU time.

For 16 processors the time with the blocked polling is almost half of it with CHT's normal polling mechanism. This is because, interrupt based blocked polling makes the effective use of CPU time therefore the calculation part of the program gets most of the time and thereby reducing the synchronization among the processors. This is case for up to 64 processors, where the calculation part is higher than the communication part. But on 128 and 256 processors where the communication part already higher than the calculation part, this effect is less obvious than the other ones. For 256 processors there is very less performance gain which suggests that the computation part of the processor is already best utilised. And as the processor increases and the computational part reduce this will not have any better effect on the performance of the application.

4.3 Conclusion

This chapter has provided various optimization strategies for the toolkit. These strategies can be implemented both on Seastar as well as Gemini Interconnects. Performance difference between the Eager and Rendezvous protocol is explained clearly along with the evident results shown in graph. More timing about these performances can be found on Appendix A.

Communication helper thread has a major effect on the performance of ARMCI library. This reason behind this is has been explained clearly. The result obtained and shown also matches the explanation. Blocked Polling mechanism and its adverse effect has also been explained with results to back it up. On the whole, all the optimizations strategies which are implemented on the toolkit have more or less a good effect on the performance.

Chapter 5

ARMCI Implementation

Two main Data transfer operation of any one sided communication is Put and Get. Even the accumulate operation is designed using these basic data transfer operation. As said in Section 4.1ARMCI's implementation of these two basic operations is different from each other. This gives us a chance to find a better way to optimize this basic operation. In this Chapter we will discuss about the Implementation of these two operations and propose a better way to optimize the ARMCI Get operation.

5.1 ARMCI Put

For Both Put and Get operations there are two types of message passing to select from, Eager and Rendezvous. More about these two variant's working and performance is explained in Section 4.1. Here we discuss about their way of implementation on Cray Systems. This is similar for both Seastar and Gemini Interconnect. The only variation is with their network level calls. ARMCI Put is already well optimized for the both the Interconnects and its working is much quicker than Get Operation. So no implementation changes are considered for the Put operation. But still its implementation is explained briefly in order to better understand the optimization proposed for ARMCI Get call.

5.1.1 Eager

ARMCI Put operation in the Eager protocol is the fastest data transfer operation in terms of initiation and completion. This is because it just uses the direct Network level Put call to transfer the data from the client to Remote Processor. Figure 5-1shows the implementation of ARMCI Put for Eager data transfer.

The client process binds the data which is to be sent with Memory descriptor (MD) of the Network level Interface. The data is then sent by a Network level Put call to the remote processor directly along with header messages in a single or multiple packets of depending on the message size restriction of the Network. The data sent by the client is then received by the Remote processors Data server (CHT) which stores the data in a buffer and sends an acknowledgement back to the client. This acknowledgement marks the completion of the ARMCI Put call on the client processor.

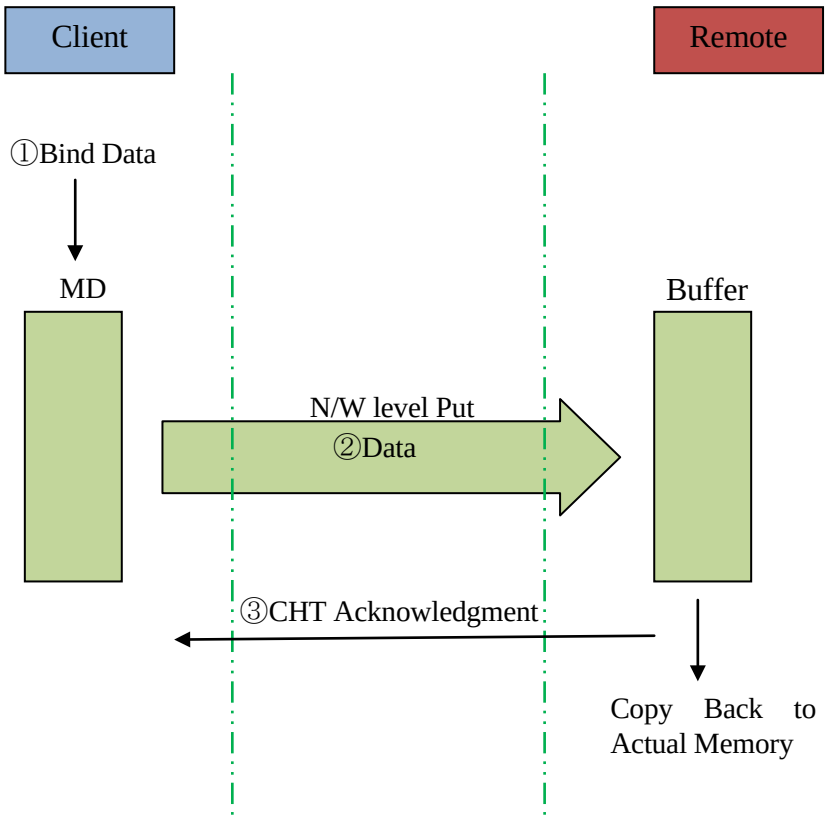


Figure 5-1 Eager Implementation of ARMCI Put

As all the one-sided communication a Memory Fence operation is needed to ensure the data transfer operation's completeness. As seen from the Figure 5-1 on the Remote Processor the data is copied back into the actual address from the temp buffer. To ensure this completeness before accessing that location a Memory Fence operation is needed. With just a single Network call and a single message transfer operation there isn't anything to optimize in the Eager version of ARMCI Put.

5.1.2 Rendezvous

Rendezvous version of ARMCI Put is very much different from its Eager Implementation. As said in Section 4.1 Rendezvous protocol is the method of transferring data only when the receiver is ready. And it also makes sure that there is no buffering and it is directly transferred to the actual address. So this involves the remote processor as well in the transfer operation, this is same for both the Seastar and Gemini Versions.

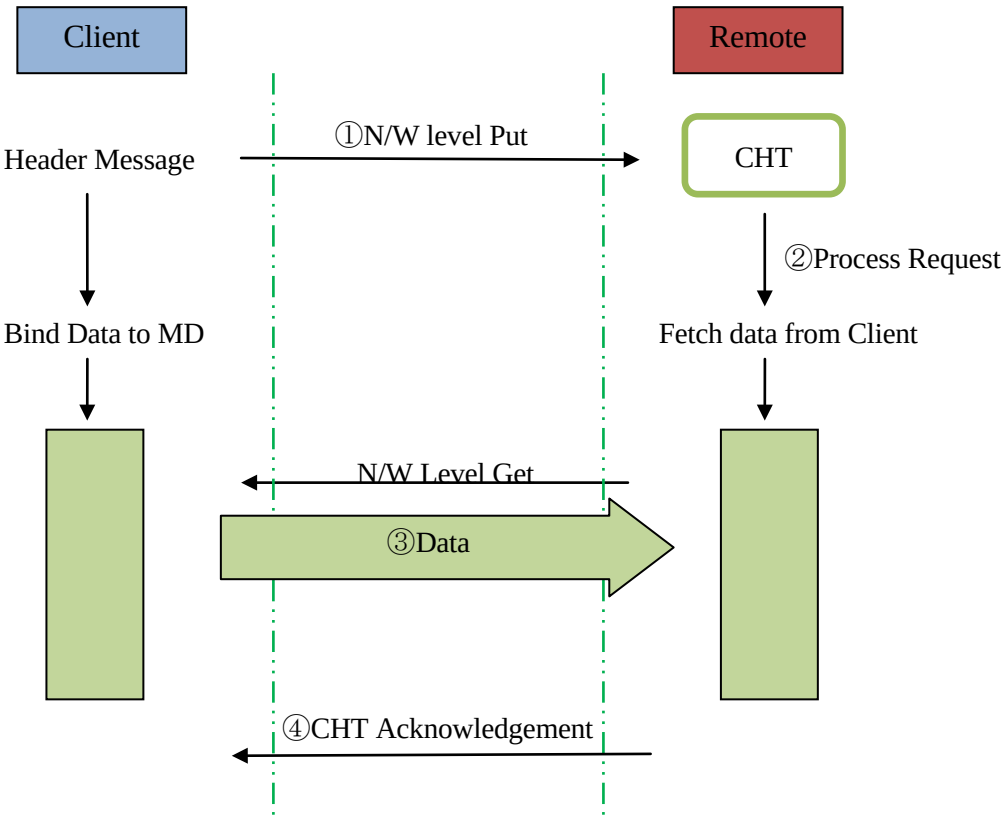


Figure 5-2 Rendezvous Implementation of ARMCI Put

As shown in Figure 5-2 the Rendezvous Put operation has more message transfers than the previous Eager implementation. Here the client processor sends a message with just the header information of the data to be sent. This message is sent using the Network Interface's Put call. Since this message size is just 96 Bytes it always falls under the Eager Put call. The Data server on the Remote Processor receives this message and it processes the request for the Put operation .Once the actual memory is attached with the Memory Descriptor, the remote processor fetches the data from the client processor by issuing a Network level Get call. The client processor on the other hand prepares the data which has to be sent and places it to for transfer in the buffer.

The Network level Get call issued by the Remote Processor identifies the data on the client processor using the match bits from the request header sent by the client. So the ARMCI Put call for rendezvous actually uses the Network Level Get call. In Portals Interface on Seastar Interconnect, Portals Get call is used to fetch the data from the client. On Gemini Interconnect the uGNI get call is used to fetch the data from the client. So the Network Level Get calls performance is what decides the performance of Rendezvous version of ARMCI's Put, not the Network level Put.

This underneath implementation is mostly unknown for most of the application programmers. So if the Network Put performance is higher than the Network's Get, then the naive expectation is that the ARMCI Put will perform better than the ARMCI Get. But it is not the case in ARMCI's implementation of message passing on Cray

Seastar as well as Gemini Network. In fact as will be seen from the next chapter this is vice versa. As said in the beginning of the chapter the ARMCI Put is implemented with best possible way for both Eager and Rendezvous Protocol, so no implementation changes are considered for ARMCI Put operation.

5.2 ARMCI Get

In the implementation of ARMCI Get on Cray XT systems there is no difference between the Eager and Rendezvous that is there is only a Rendezvous implementation. Therefore the Get calls performance does not vary depending on the buffer size, which is explained in Section 4.1.2. In this section we analyze the current implementation of the ARMCI Get call and propose an alternative implementation for the Eager version of ARMCI Get. Implementing this proposed version of ARMCI Get is beyond the scope of this project. But a detailed approach for implementing this is discussed here.

5.2.1 Current Implementation

Present implementation of ARMCI Get call involves both the client and remote processor for the data transfer. It follows Rendezvous protocol method, this means there is no use of buffers or there is no buffering of data during the transfer.

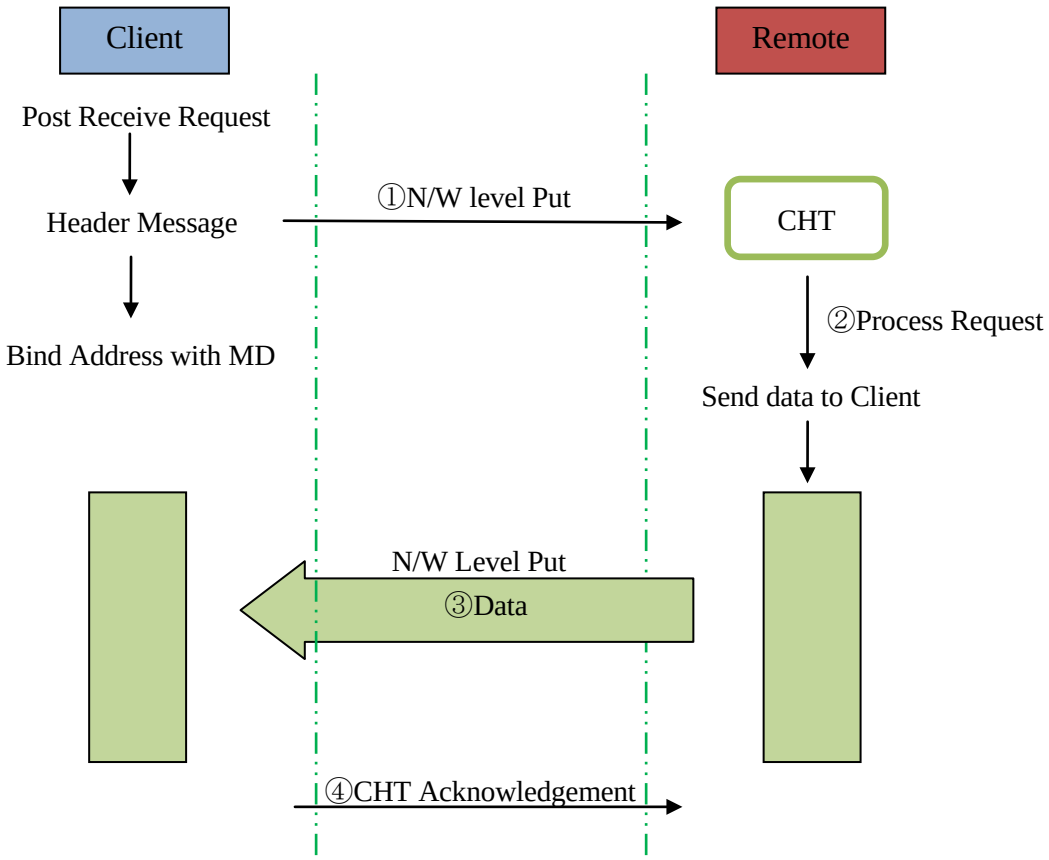


Figure 5-3 Present implementation of ARMCI Get on both Seastar and Gemini

Figure 5-3 shows the current implementation of the ARMCI Get call. In this the Client processor pre posts a request to the receive queue before initiating the communication and associates the actual memory address to Memory Descriptor. So now the data can now directly go into the actual address without the need of buffering. Once this is done the client processor sends a request to the remote processor with the header message of the data needed by it. This request is done by sending the message using a Network level Put call. As in the Rendezvous version of ARMCI Put call (Section 5.1.2) this request message is also of size 96 Bytes and falls into the Eager Put version.

Once the Data Server on the Remote processor receives this request, it processes the request to check the nature of request. And once it finds out the request for data, it initiates a data transfer operation. Before sending any data back, the Data Server processes the requested data and packs it based on the contiguous or Strided request and transfers it to the Client processor using a Network level Put operation. So in simple words, the client processor sends a message requesting the data and the remote processor sends the data back to the client processor.

It is very clear from the Figure 5-3, that ARMCI Get does not use any Network level Get calls. So the performance of ARMCI Get is completely dependent on the Performance of Network Interface Put operation. This kind of implementation is certainly good when the size of the data is high, so the initial cost of request message can be neglected when considering the whole data transfer operation. For non contiguous Strided messages involving the Data Server of the remote processor helps in packing the data into single contiguous message. And also there is no buffering on either processor which reduces the memory used and helps in the case of memory intensive applications. But for a small message the total cost of the message transfer will be twice that of the ARMCI Put, which means the latency cost of Get is very higher than ARMCI Put.

5.2.2 Proposed Implementation

As said in the previous section the latency cost of the ARMCI Get is very higher even for the small messages. So a different method of implementation for Eager Version of ARMCI Get is proposed. For the data of size less than the threshold limit this type of data transfer operation can be used.

This implementation is similar to the Eager version of the ARMCI Put. Here the client processor alone involves in the message passing and the remote processor is not directly involved in the process. Since the actual memory where the data has to be stored can be attached to the memory descriptor before the communication, this method does not involve buffering as well.

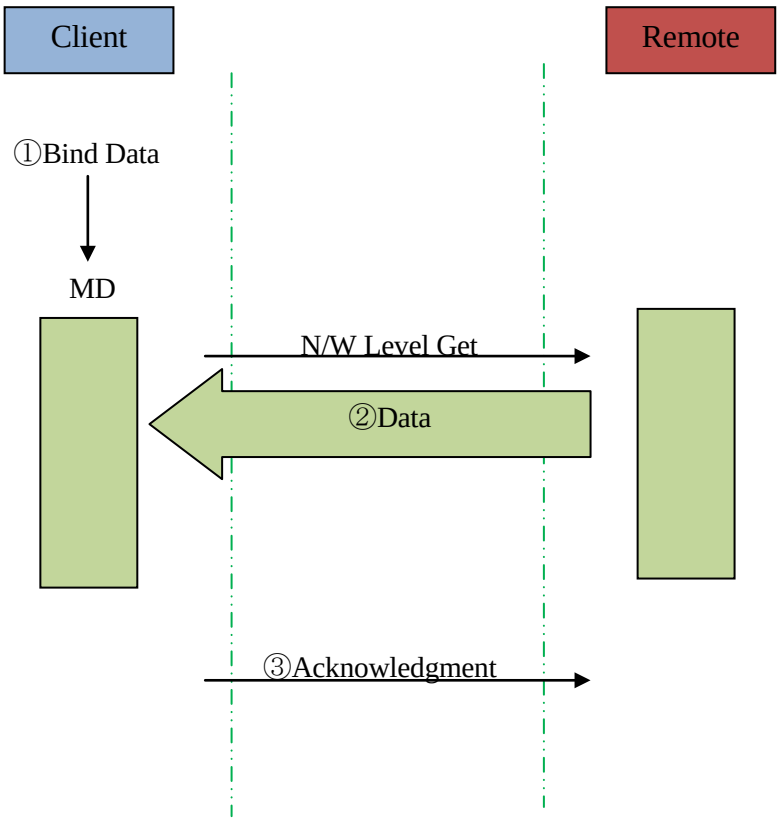


Figure 5-4 Proposed Implementation for Eager Version of ARMCI Get

The client processor attaches the actual address to the Memory descriptor of the Network Interface. Once the memory descriptor is prepared, the client processor issues a Network level Get call to the remote processor (Figure 5-4). This Network level get call directly fetches the required data from the remote processor, without the involvement of the Data server of the remote processor. Since there actual address of the memory on the client is attached to the Network Interface's Memory descriptor, the fetched data directly stored there. This results no buffering hence saving most of the time.

This method can be implemented for both blocking and Non-Blocking versions of the Get call. The Network level one sided Get call is a non blocking call by itself. So using a non blocking Network level get call helps in implementing for non blocking version and the blocking version can be accomplished by including a Network level wait call along with it. The acknowledgement mentioned in the figure is an optional one, since the remote processor is not actively participating in this and also there is no involvement of the data server to intimate the completion of the operation.

In Gemini Interconnect with the support of global addressing, uGNI Get call can directly the access the memory of the remote processor. So this type of implementation will reduce the latency cost very much than the present implementation. This is

possible in Seastar Interconnect as well. Seastar does not support global addressing so there is no way to access the remote data directly without the involvement of the remote processor. But the Portals Get function in the Seastar Interconnect is by itself a two sided message. The Portal Get sends a request to the remote interconnect and the interconnect accesses the data on the remote memory by itself or with the help of remote processor for calculating the actual address using the Portals Match bits. So this method of ARMCI Get call is possible to implement on both the Interconnects.

Limitations

This type of implementation has a few limitations in it. This method when applied to a large message may flood the network with many messages. Since there is no Data server (CHT) involved in the remote processor, the messages are not packed or sent in packets of small sizes to fit the Network Interface. Direct fetching of data involves only he interconnect, so it cannot pack or divide the data. So this checking can be done in the client processor and this method can be used for message size which is less than the Eager cut off limit or a single packet size of the Network.

As well as for the Strided Non contiguous data, this method may not perform better as it performs for contiguous data. For Strided data this direct fetching cannot be done and hence a complete data will be fetched, which will transfer a large amount of unnecessary data. In the current implementation the striding of the data is done by the Data server on the Remote processor (Section 5.2.1). So the lack of the involvement of Data server means no striding is possible. One possibility is that, the total data can be fetched from the remote processor to the client and the striding can be done in the client. But this involves transferring large unwanted data, which is not feasible. So this implementation can be used for the Contiguous messages less than the Eager limit.

5.3 Conclusion

The variation in the implementation of the ARMCI Put and ARMCI Get, provides us the opportunity to consider various type of implementations. Current implementation of these operations in the Seastar and Gemini Interconnect (which is same) is explained. The difference in the Get variation and its drawback has been addressed.

An alternative implementation for the Eager version of Get operation is proposed. Implementing this is way beyond the scope of this project. None the less, all the implementation details and the expected results on the toolkit are also analyzed. The drawbacks expected on this new proposed implementation are also discussed.

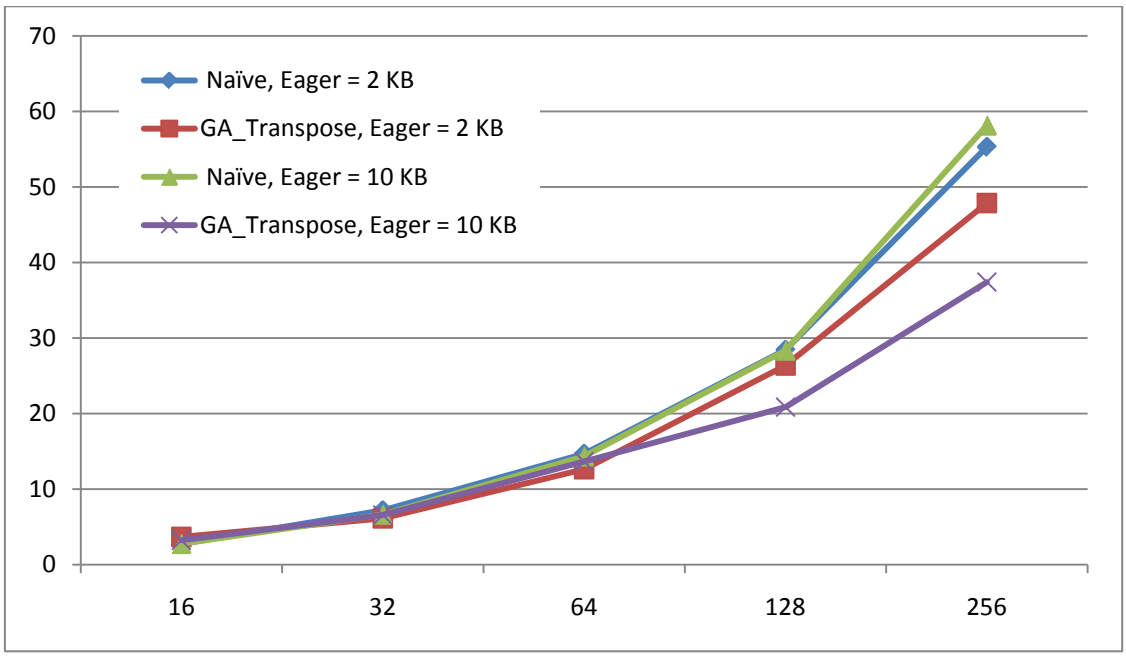
Chapter 6

Results and Discussion

The results obtained for all the analysis and optimizations done is explained throughout the report. Here a collective result of all the optimizations done is discussed. Two different codes are considered for benchmarking and analyzing the Global Array toolkit. Results obtained on both these codes are given below.

6.1 Matrix Transpose

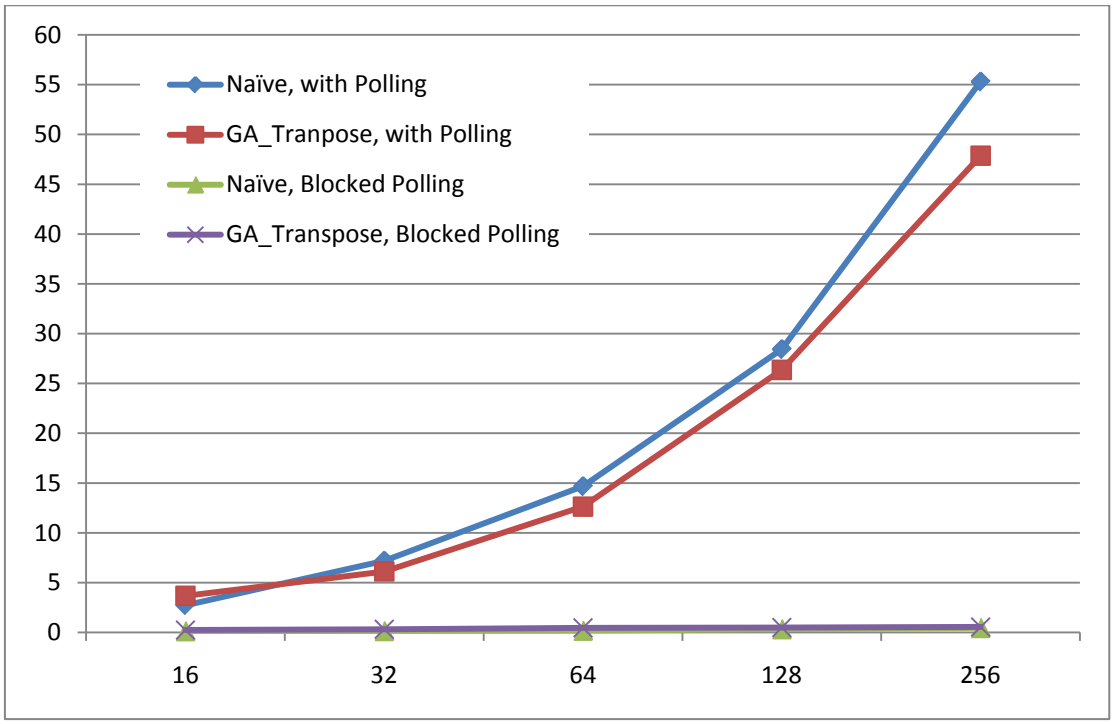
Since Matrix transpose benchmark has no calculation part, the optimizations done on the Toolkit's communication part certainly yields a good performance gain. Changing the communication pattern to Eager from Rendezvous gives a good increase in performance for the GA_Put operation as the GA_Transpose version of the implementation runs quicker than the naive implementation.



Graph 6-1 Showing Naive and GA_Transpose implementation for Eager Cut off limit of 2 KB and 10 KB

As seen from the Graph 6-1 the Naive implementations of the Matrix transpose doesn't get any performance gain, this is due to the lack of Eager implementation for ARMCI Get, which is explained in Section 5.2. But the GA_Transpose call gains a good performance gain as the number of processor increases; this is due to the lower message size and less synchronization time.

But when using the Blocked polling of the Communication Helper thread, the time will drop down completely. But it is not a good measure to use this Matrix transpose with CHT blocked polling. Since Matrix transpose just has the communication part, there will in fact no CPU time needed and therefore the blocked polling will perform way faster than the normal ones.



Graph 6-2 Showing Matrix transpose with and without Blocked Polling

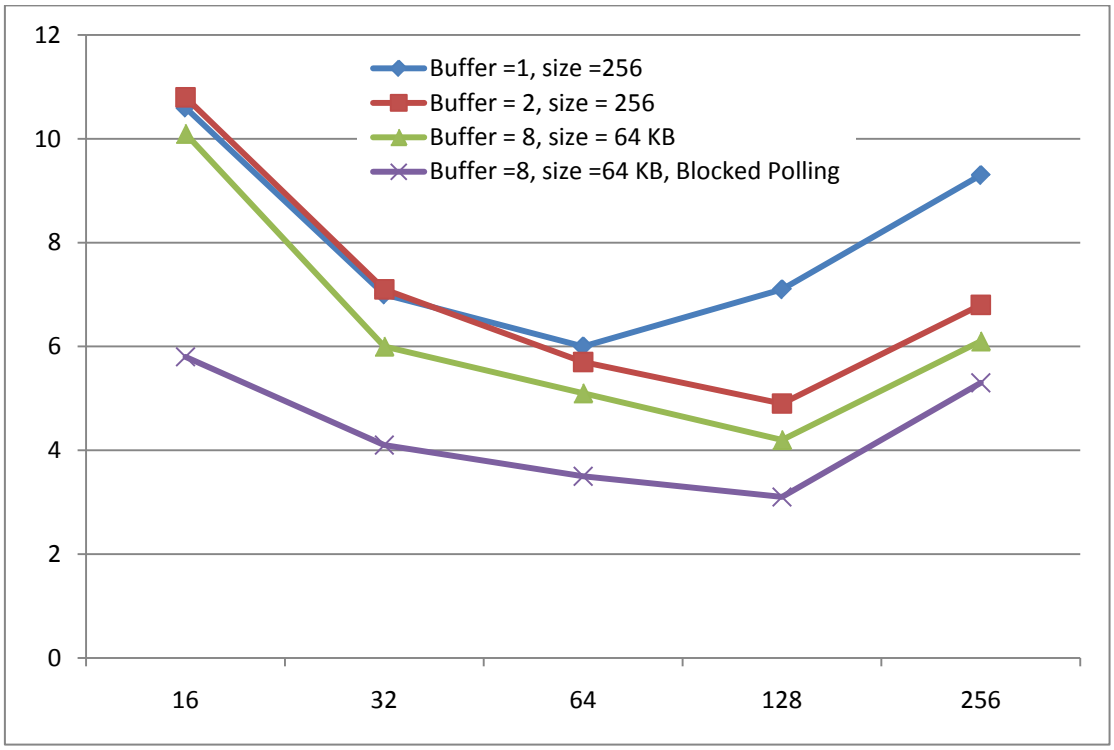
As seen in Graph 6-2 the timing for the Matrix transpose operation with blocked polling is not even a considerable one. This is due to the fact that the thread is completely blocked when its free of any receive queue and there is very less or in fact no synchronization time needed. So it is not a good measure to use such a calculation free Matrix transpose operation for analysing the Toolkit by using a Blocked polling option in CHT. There is only one global Get, in case of Naive implementation and one global Put in case of GA_Transpose on every processor. So after this the entire thread is blocked and so the synchronization is already done with the Global Data transfer call. So the fence operation does not take any time as they are already in the same state.

With the blocked polling, the CHT thread continues to run and poll for all the messages from the processors leaving no CPU time on the processor which runs the CHT on each node. And this thread continues to poll even after receiving the entire set of data from

all the processor expecting more data or waiting for its completion, which further increases the synchronization time among the processors on the node. Thereby the increasing whole synchronization time, this grows higher and higher as the number of processor increases. This accounts to the huge amount of difference in time for the method with blocked polling and the one without blocked polling.

6.2 NWChem

NWChem module's extensive use of Get and Put calls with its more memory usage helps in understanding and analyzing the performance of the toolkit for real time application.



Graph 6-3 Showing NWChem DFT Modules Performance for various optimizations

The performance of NWChem has gradually increased for all the optimizations done. Graph 6-3 shows the performance increase attained in the NWChem DFT Module, which is considered for benchmarking. The GA Statistics for this module is shown in Table 3-1. With a large number of Get and Put calls, these communications optimizations are sure to produce good performance optimizations.

In the Graph 6-3 the performance is bad when the number of CHT buffers available is set to the minimum of 1 buffer per client. As the number of increases the performance gradually increases. But this also brings in the complexity of more memory usage for buffers, which will affect the memory intensive applications (Section 4.2.1.1). So increasing the number of buffers and decreasing each buffer size will balance this

memory problem. So even for memory intensive applications the number of buffers can be increased with the reduced buffer size, which also yields a good performance gain as shown in the graph with green line. But it is the use of CHT blocked polling which gives more performance gain than the effective buffer utilization (Section 4.2.2).

Using the blocked polling for CHT saves the CPU time considerably from polling and waiting on the network events. This speeds up the calculations by effectively using the CPU time. This also reduces the synchronization time of the processor since no processor will be sitting idle at any time even if there are no network events in the queue. It also provides fair communication opportunity for all the processor on the nodes by allowing them to send messages asynchronously even if there are many number of messages waiting in the receive queue. This helps in the performance gain of the application. But this effect reduces as the number of processor increases where there is very less calculation and only synchronization, this is clear from the Graph 6-3. For 256 processors all the gap between the previous non blocked version and the blocked polling version seems to reduce. So this indicates, further increasing the processor count will still reduce this gap finally having no effect on the performance.

Chapter 7

Conclusions

Global Array toolkit is entirely dependent on the ARMCI one sided communication library for all its communication. Analyzing and Optimizing the ARMCI's implementation on Cray XT systems for Seastar Interconnect and Gemini Interconnect is the main course of the project. As Per EPCC's agreement with The Cray Supercomputing Company, the results or performance measures of the Gemini Interconnect cannot be published at the moment. So these results are with held under as per the agreement within the EPCC's internal document. If in case of needing the performance results of the Gemini System, it can be obtained by contacting my supervisor Dr. Andrew Turner, EPCC.

Many Performance optimization techniques have been analyzed throughout the report. Buffer utilization proved to be very important and effective usage of the buffers results in very performance gain. The difference in the performance obtained with the effective usage of the buffer is explained in section 4.2.1, this result is very good on the Seastar Interconnect. And the initial result obtained from the Gemini Interconnect is also encouraging.

Message transfer protocol has a tremendous effect on the ARMCI Put call, which is explained in section 4.1. So deciding on the Eager Cut off limit becomes a decisive factor and it hugely depends on the memory available on the system and its rate of copying from buffer to actual memory. The limitations and performance gains are also analysed and discussed in that section.

ARMCI's lack of Eager version of Get call, affects the latency of the Get calls. This in turn will affect the performance of the small messages which are fetched by the client processor from the remote processors. So a different version from the current implementation of the ARMCI Get on Cray Systems is proposed. This by measuring the performance gain obtained from Eager version of Put from the rendezvous version of put seems promising. This method can be implemented on both Seastar and Gemini Interconnects.

Optimizing Global Array Toolkit ultimately comes down to optimizing the ARMCI one-sided library. And ARMCI by itself is very well optimized library, so obtaining major effects on the performance of the ARMCI library is a very big task. But None the less many versatile techniques have been analysed and few interesting methods have been proposed. As seen throughout the report all the results have shown very good effect in Seastar Interconnect. And due to the legal rights the results obtained on Gemini Interconnect can be published. But many test cases and runs have to execute on the Gemini Interconnect and the results obtained from these test runs seems encouraging.

7.1 Further Work

Although most of the available optimizations have been analysed and implemented in this project. There is always a better scope for improvement in any field. With Gemini Interconnect just out to the market there are various things which can be further done to this project.

As said in Section 5.2.2, this method of Eager version for ARMCI Get is just designed and proposed in this project. And this cannot be implemented in the short duration of the project. So implementing this proposed design is a huge step forward in optimizing the Get call. This will surely have very good effect on Gemini systems since the hardware supports global addressing directly.

With more communication channels and bandwidth on the Gemini Interconnect, a better global operation method can be designed to effectively use the high bandwidth. Gemini is proposed to have very impact on the MPI_Alltoall communication, so this means it can serve a very high load of messages on the network and the latency time will reduce as well. So these are all very good signs for implementing a different global one sided communication call.

7.2 Post-mortem

The risk analysis done during the project showed the possible non availability of Gemini System. And since the risk was expected previously it has been effectively overcome. The codes and test cases for the Gemini System has handed over to my supervisor Dr. Andrew turner, who in turn had run the test cases on my behalf and returned me with the results.

On the whole the project has generally met all the desired objectives proposed during the project preparation. It has been able to come with a detailed analysis of the Global Array Toolkit and thee by optimizing it for better performance on Cray systems. Even though the results obtained on the Gemini system can be published, all the explanations given in the report stands well for Gemini Interconnect as well.

Appendix A

Matrix Transpose timings

On Phase2a

Naive Implementation for 6400 x 6400 Integers, Eager Limit = 2KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.2306	6.4044	2.7317
32	4.5002	10.6802	7.1907
64	12.797	16.3681	14.6724
128	26.2139	31.7896	28.4385
256	51.9188	57.8035	55.3099

GA_Transpose Implementation for 6400 x 6400 Integers, Eager Limit = 2KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.638	6.3524	3.6794
32	3.4	8.374	6.1224
64	10.8233	14.7009	12.6315
128	23.1769	30.2018	26.3648
256	44.1398	52.1051	47.867

Naive Implementation on 12800 x 12800 Integers, Eager Limit = 2KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.482667	4.052648	2.889232
32	4.493685	8.126387	6.153598
64	11.864198	19.27909	15.167708
128	28.201501	32.10569	30.318007
256	52.903287	61.512601	57.334704

GA_Transpose Implementation for 12800 x 12800 Integers, Eager Limit = 2KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	2.500044	6.912718	4.351654
32	4.351654	4.351654	6.619428
64	12.231743	15.069272	13.393149
128	23.921589	26.792148	25.122136
256	44.184603	49.497085	46.185113

Naive Implementation 6400 x 6400, Eager limit = 10 KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.0998	4.4047	2.7654
32	4.7525	9.0237	6.644
64	12.568	17.3538	14.3456
128	24.22	32.0243	28.3523
256	52.2102	56.5964	55.1553

GA_Transpose Implementation for 6400 x 6400 Integers, Eager Limit = 10KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.8998	5.4002	3.2181
32	4.7015	9.5791	6.574
64	12.588	18.0558	13.6426
128	18.6536	23.8487	20.8535
256	35.3542	40.6544	37.4127

Naive Implementation 6400 x 6400, Eager limit = 42 KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.0998	4.4047	2.7654
32	4.7525	9.0237	6.644
64	10.07004	18.901077	14.8146
128	24.22	32.0243	28.3523
256	52.2102	56.5964	55.1553

GA_Transpose Implementation for 6400 x 6400 Integers, Eager Limit = 42KB

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	1.8998	5.4002	3.2181
32	4.7015	9.5791	6.574
64	9.013689	12.2824	10.66
128	18.6536	23.8487	20.8535
256	35.3542	40.6544	37.4127

Naive Implementation 6400 x 6400, with Blocked Polling

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	0.083012	0.249833	0.13951
32	0.093896	0.273836	0.152705
64	0.084229	0.279476	0.188451
128	0.267711	0.50697	0.334966
256	0.363368	0.576137	0.454862

GA_Transpose Implementation 6400 x 6400, with Blocked Polling

Processor (Cores)	Min Time (in secs)	Max Time (in secs)	Avg time (in secs)
16	0.143947	0.554767	0.242008
32	0.071948	0.650147	0.320622
64	0.179835	0.726143	0.46349
128	0.209966	0.957602	0.491521
256	0.346745	0.896344	0.548501

On Phase 2b

Array size of 9600 x 9600 integers (Time in secs)

Processor (Cores)	Naive Implementation	GA_Transpose Implementation	GA_Get
24	0.25	0.3	0.02
48	0.34	0.37	0.29
96	0.6	0.49	0.34
192	1.05	0.97	0.83
384	2.39	2.1	1.93

References

1. Cray XT™ System Overview. [Online] December 2009. [Cited: 26 August 2010.] http://docs.cray.com/books/S-2446-31/S-2446-31.pdf.

2. HECToR Hardware. [Online] HECToR, UK National Supercomputer. [Cited: 27 August 2010.] http://www.hector.ac.uk/support/documentation/userguide/hardware.php#xt5h.

3. **Sheets, Kitrick.** *High-Speed Network Communications on the Cray XT.* s.l. : The Cray Supercomputing company.

4. **Cray Inc.** *The Gemini Network Rev 1.1 (White Paper).* s.l. : Cray, The Super computing company, 2010.

5. **Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease.** *Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit.* s.l. : Pacific Northwest National Laboratory.

6. *Global arrays: a portable "shared-memory" programming model for distributed memory computers.* **J Nieplocha, R J Harrison, R J Littlefield.** 1994, ACM/IEEE Conference Supercomputing.

7. Global Array Toolkit. [Online] Pacific National Laboratory, U.S. [Cited: 27 August 2010.] http://www.emsl.pnl.gov/docs/global/.

8. ARMCI. [Online] Pacific National laboratory. [Cited: 26 August 2010.] http://www.emsl.pnl.gov/docs/parsoft/armci/.

9. ARMCI Project status. *ARMCI.* [Online] [Cited: 24 march 2010.] http://www.emsl.pnl.gov/docs/parsoft/armci/index.html#project_status.

10. **Rolf Riesen, Ron Brightwell, and Kevin Pedretti, Sandia National Laboratories.** *The Portals 3.3 Message Passing Interface 2.1.* Albuquerque, New Mexico 87185 and Livermore, California 94550 : Sandia National Laboratories, 1999.

11. Using the GNI and DMAPP APIs. [Online] june 2010. [Cited: 26 August 2010.] http://docs.cray.com/books/S-2446-31/S-2446-31.pdf.

12. ARMCI Performance. *ARMCI.* [Online] [Cited: 24 march 2010.] http://www.emsl.pnl.gov/docs/parsoft/armci/performance.htm.

13. **Vinod Tipparaju, Edoardo Aprà, Weikuan Yu, Jeffrey S. Vetter.** *Enabling a highly-scalable global address space model for.* Bertinoro, Italy : s.n., May,2010.

14. NWChem. *NWChem.* [Online] [Cited: 24 march 2010.] http://www.emsl.pnl.gov/capabilities/computing/nwchem/.

15. Density Functionals from the Truhlar Group. [Online] [Cited: 26 August 2010.] http://comp.chem.umn.edu/info/dft.htm.

16. **Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer, Vinod Tipparaju.** Global Arrays User Manual. *Global Arrays Website.* [Online] july 2009. [Cited: 24 march 2010.] http://www.emsl.pnl.gov/docs/global/user.html.

17. *High Performance Remote Memory Access Communication: The Armci Approach.* **J. Nieplocha, V. Tipparaju, M. Krishnan, D. K. Panda.** 2, s.l. : International Journal of High Performance Computing Applications, 2006, Vol. 20. 233/253.

18. NWChem. [Online] Department of Energy EMSL, U.S. [Cited: 27 August 2010.] http://www.emsl.pnl.gov/capabilities/computing/nwchem/.