A Micro-Benchmark Suite for Unified Parallel C

Sainath Lakshminarayanan

August 21, 2009

MSc in High Performance Computing
The University Of Edinburgh
Year of Presentation: 2009
Abstract

As the computing power of supercomputers is increasing at an exorbitant rate every year, the need for a programming paradigm to fully utilize their potential becomes vital. Partitioned Global Address Space, a new class of programming model, tries to address this issue with its ease of programming and effective memory layout.

This report looks into the performance of one of the PGAS languages, Unified Parallel C. The aim of this project was to design and implement a micro-benchmark suite which will be used to find the overheads associated with various programming constructs of UPC. Cray XT4 and X2 systems were used to carry out this project and a Sun x4600 machine was used for testing purposes. Performance of two popular implementations - Berkeley UPC and Cray UPC - were analysed on the two systems using the micro-benchmark suite. A general comparison of the two implementations based on their performance has been presented in this report.
# Table of Contents

1 Introduction..................................................................................................................... 1

2 Background...................................................................................................................... 6
   2.1 Partitioned Global Address Space Languages ..................................... 6
   2.2 Unified Parallel C ........................................................................................... 8
      2.2.1 Overview ............................................................................................. 8
      2.2.2 Language features ........................................................................... 8

2.3 Benchmarking ..................................................................................................... 18
   2.3.1 Challenges involved with Benchmarking ................................... 18
   2.3.2 Different Types of Benchmarking.......................................... 19

2.4 Existing Benchmarks ................................................................................ 20
   2.4.1 GWU Benchmarks ......................................................................... 20
   2.4.2 Berkeley Benchmarks ................................................................. 21

2.5 Motivation ..................................................................................................... 23

3 Design & Implementation ....................................................................................... 24
   3.1 Design of Timing Routine ........................................................................ 24
   3.2 Upc_forall ....................................................................................................... 25
      3.2.1 Use of a dummy routine............................................................. 26
      3.2.2 Use of single variable update ................................................... 26

   3.3 Synchronisation Constructs ................................................................... 27
      3.3.1 Barrier ............................................................................................... 28
      3.3.2 Notify and Wait .............................................................................. 28
      3.3.3 Lock and Unlock ............................................................................ 29

   3.4 Collective Operations ................................................................................. 30

   3.5 Implementation details ............................................................................. 30
# TABLE OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Memory layout in UPC</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Cyclic distribution</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Blocked cyclic distribution</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Indefinite block</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Timing a section of code</td>
<td>25</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Data structure for contiguous allocation</td>
<td>34</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Performance of broadcast on Cray XT4 (Berkeley implementation)</td>
<td>37</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Performance of broadcast on Cray X2 (Cray implementation)</td>
<td>37</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Time vs. Processors for broadcast operation (16384 words)</td>
<td>38</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Performance of scatter on Cray XT4 (Berkeley implementation)</td>
<td>40</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Performance of scatter on Cray X2 (Cray implementation)</td>
<td>40</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Time vs. Processors for scatter operation (16384 words)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Performance of gather on Cray XT4 (Berkeley implementation)</td>
<td>41</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Performance of gather on Cray X2 (Cray implementation)</td>
<td>42</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Time vs. Processors for gather operation (16384 words)</td>
<td>42</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Performance of allgather on Cray XT4 (Berkeley implementation)</td>
<td>44</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Performance of allgather on Cray X2 (Cray implementation)</td>
<td>44</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Time vs. Processor for allgather operation (16384 words)</td>
<td>45</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Performance of exchange on Cray XT4 (Berkeley implementation)</td>
<td>45</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Performance of exchange on Cray X2 (Cray implementation)</td>
<td>46</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Time vs. Processors for exchange operation (512 words)</td>
<td>46</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Performance of permute on Cray XT4 (Berkeley implementation)</td>
<td>48</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Performance of permute on Cray X2 (Cray implementation)</td>
<td>48</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Time vs. Processors for permute operation (16384 words)</td>
<td>49</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Performance of reduce on XT4 (Berkeley implementation)</td>
<td>49</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Performance of reduce on X2 (Cray implementation)</td>
<td>50</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Time vs. Processors for reduce operation (16384 words)</td>
<td>50</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Lock and unlock on Cray X2 and XT4</td>
<td>52</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Barrier on Cray X2 and XT4</td>
<td>53</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Notify and wait on Cray XT4 and X2</td>
<td>53</td>
</tr>
</tbody>
</table>
Acknowledgements

I sincerely thank Dr. Mark Bull for suggesting this project and also for his invaluable guidance and support in all phases of the project.

I would like to thank Dr. Alan Gray for his assistance during the project. I thank the Berkeley UPC mailing list, especially Gary Funck and Paul H. Hargrove for their technical assistance in UPC. I thank Savvas Petrou who helped me with the installation of UPC initially.

I am grateful to Sreelakshmi for taking her time to proof read this work. Special thanks to BBKKS group for their undaunted support throughout the course. Last but not the least; I thank my parents for helping me pursue this course.
1 Introduction

In the past 15 years we have seen a fast and steady growth in the field of High Performance Computing. The processing capacity of the fastest supercomputer reached a peak performance of a teraflop in 1997, and 11 years later, a petaflop in 2008. In 2008 with the help of accelerators and quad core processors, the first petaflop supercomputer was built. Research has already begun in building an exaflop supercomputer and is expected to be developed by 2020.

The increase in productivity of hardware means that there has to be an increase in productivity of software as well to harness its power. If we look at software history, a formalised standard for programming supercomputers called Message Passing Interface (MPI) [1] was created in 1994. This is, to date, the de-facto standard for programming supercomputers with distributed memory architecture.

Message Passing Interface is a specification for communication between processes in a distributed system. It is also the most widely used communication interface for supercomputers and clusters. The disadvantage of MPI is that it is very hard to develop a parallel application from an existing non-parallel code as the programming model is complex. For this reason it is often called as the assembly language for parallel processing [2]. However, for scalable architectures this is the only popular solution that is available.

In 1997 an API for shared memory systems, OpenMP was developed [3]. The programming model of OpenMP is very simple to learn and use. OpenMP became the de-facto standard for programming parallel applications in shared memory systems as it is very easy to parallelize existing serial code. Implementation of OpenMP in distributed memory systems is very hard and even if developed, its performance would be poor compared to MPI.

The increase in number of processors affects the performance of MPI. The current MPI specification may not be able to fully utilize the processing capabilities of supercomputers with hundreds of thousands of cores. There are many issues involved in using MPI for such massively parallel architectures. These include, but are not limited to scalability, message buffering and replicated data.
A detailed work on the performance issues with respect to scalability of MPI has been discussed in [4]. This work addresses the scalability issues, with the implementation of MPI specification and with the MPI specification itself. In this work the issues in the specification, like the irregular collectives which allow unequal transfer of message size, graph topology, one sided communication of MPI, All-to-All communication, representation of processor ranks and fault tolerance, were discussed. These are some of the non scalable aspects of the specification and so even the best implementation of these functions and representations will not be scalable. Implementation issues like effective process mappings, memory overheads in communicator creation, scalability of MPI_Init, optimization of collective algorithms were also discussed in this work.

The performance of MPI is also dependent on message buffering. Message buffering is the storage of data, between a send and the corresponding receive of that particular data. If the number of processors increase and if the program involves communication between many processes message buffering becomes an important issue as buffer space available is finite.

In the MPI model, during program initialization some number of processes, say N are created and generally, N equals the number of cores used by the program. Each process communicates via sending messages explicitly. Synchronisation is implicit in the message passing model. This is so because communications take place with a send and a matching receive. Within a node having many cores explicit message passing will not be required as we could simply copy data within the memory which is being shared by all the cores in that node. In a node with many cores, having explicit synchronisation constructs rather than associating synchronisation with communication is a better alternative, as separating synchronisation and data transfers may improve bandwidth.

In the future when machines with a million processors come into existence, as pointed out in [4] it is highly unlikely that applications will treat all the MPI processes in the same way. The memory available per individual core will decrease and also due to increase in systems size, effect of message buffering will become more pronounced. Therefore usage of shared memory model within nodes will be encouraged, especially if we have hundreds of cores per node.

One more important thing to consider is that in the near future we may not just have CPUs in the supercomputers. We may have other functional units, such as GPUs, as well. The MPI model might not be able to effectively make use of the capabilities of other functional units apart from CPUs.

An obvious alternative to this problem might be the use of a hybrid model consisting of MPI and OpenMP. But such a hybrid model will also have many limitations. MPI and OpenMP are two different programming models with two different approaches and so they might not blend together properly. OpenMP model has alternating parallel and serial sections and hence the speed of OpenMP section will be limited by the serial section of
the code according to Amdahl’s law [5]. Programming in hybrid model is also more complex than programming in MPI or OpenMP models separately.

To overcome the programmability issues of MPI and Hybrid model discussed previously, and to make programming easier data parallel languages came into existence. One of the most famous data parallel languages, High Performance Fortran (HPF) was developed in 1993. The programming model of data parallel language is very easy to understand. The time taken to develop a parallel application from a serial code in a data parallel language will take less effort than in MPI.

The data parallel model, however, had some issues which made it unsuitable for large supercomputers. Communication cost, which affects the performance of data parallel programs, is dependent on factors like data alignment, mappings, distribution, array operations etc. It is difficult to develop programs that effectively take care of all these factors. Many important details like data locality are abstracted away from the user and hence the performance of the program depends not only on the programmer but also on the implementation of the compiler.

Another interesting programming model to look at is SHMEM, which is similar to the message passing model but different only in that the communication between processes is a one step process unlike in MPI where it is a two step process. Data transfer requires only one process to read or write directly from the remote processor’s memory. The CPU of the remote processor is not interrupted during data read or data write operations. The remote processor does not know if its memory has been read or written by any other processor. It is the responsibility of the programmer to make sure that the remote processor knows this. The SHMEM model is widely used only by Cray. It is tightly coupled with the architecture of the system and so the performance of SHMEM depends largely on the underlying hardware. Despite its performance, due to portability reasons it is not a popular programming model.

The alternatives suggested for MPI and hybrid model had many problems, some of which we saw, which made them unsuitable for programming high end supercomputers. In order to prepare ourselves for the exaflop era active search for better alternatives, which might address all the issues related to using MPI, hybrid model or other programming models discussed above, began.

To cope up with the problem of finding an efficient programming language for the next generation of supercomputers, Partitioned Global Address Space (PGAS) languages were developed as part of the High Productivity Computer Systems (HPCS) [6] project started by DARPA. These languages were developed to address some of the key issues like

- Performance
- Ease of Programmability
- Portability
• Reliability (Robustness)

The advantages of PGAS model are

• It uses single sided communication which is very effective for programs having an unstructured communication pattern. For example, while modelling a Molecular Dynamics problem we will not know in the start the communication pattern that will be followed throughout the program. In that case, based on performance, using PGAS languages would be better than MPI
• In terms of programmability, it is closer to the OpenMP programming model, which is easy for developing parallel applications
• It has a natural fit for modern distributed memory systems as data locality information is extensively used by this model
• If supporting hardware is available then single sided communication used in PGAS model, reduces communication overhead and hence increases scalability and performance

Although the new MPI 2.0 specification [7] supports single sided communication it imposes many restrictions on the memory access pattern. A detailed work on some of the shortcomings of MPI 2 and also explanation as to why it cannot be used as a compilation target for parallel languages is discussed in [8].

Many vendors, funded by HPCS, started coming up with their own proprietary languages based on this new model. Cray developed Chapel [9], Sun Microsystems came up with Fortress [10] and IBM with X10 [11]. But all these languages were tailored to suit the manufacturer’s hardware. Every time a new supercomputer from a specific vendor is bought, it is essential that the users learn the programming language developed by that vendor.

In order to avoid this, a different set of languages were developed which could run on all the available platforms. There are three PGAS languages available in this line. They are Co-array Fortran, Unified parallel C and Titanium.

The rest of this report will look closely at one of the PGAS languages, Unified parallel C, and its performance in terms of the overheads associated with the programming constructs of the language. This is done with the help of a micro-benchmark suite developed specifically for this purpose.

The purpose of this micro-benchmark suite is to analyse the performance of the different compilers available for UPC. The micro-benchmark suite will help in identifying constructs that needs to be implemented in a better way for a particular compiler.
This report has been structured in the following way.

Chapter 1 gives a brief overview about the popular standards used in the HPC industry. It also discusses the issues related to the already existing standards. Finally, the alternative languages which are recently developed to address the problems related to the old standards are mentioned.

Chapter 2 gives a brief overview of PGAS languages. Section 2.1 discusses the general features of PGAS languages. Section 2.2 gives specific information on Unified Parallel C. Most of the features with respect to commands available in UPC - which are relevant to this project - are discussed in this section. Section 2.3 gives a brief overview of the different types of computing benchmarks that are in use and also the challenges involved in using benchmarks. In Section 2.4 we look at all the existing benchmarks for UPC. We speak about what these benchmarks have achieved and what is missing. In section 2.5 we discuss the motivation behind this work and also the proposed outcome of this benchmark.

In Chapter 3 we discuss about design and implementation of the micro-benchmark suite. Section 3.1 discusses about the timing routine used in the program. Section 3.2 looks at the design of the benchmark for upc_forall loop. In Section 3.3 we discuss the design of the benchmark for acquiring and releasing locks, barriers and split-phase barriers. Section 3.4 discusses in detail the design for testing the overheads associated with the collective routines. Section 3.5 gives all the implementation details, including information on how to use the benchmark, the data structures used.

In Chapter 4 we present the results of running the benchmark on two different compilers. The compilers that were used are Cray’s UPC compiler and Berkeley’s UPC compiler. The benchmark has been run on Cray XT4 and X2 machines. In Section 4.1 a brief overview about the architecture of the two machines used is given. Sections 4.2, 4.3 and 4.4 discuss the results of executing the benchmark for collective operations, synchronisation constructs and upc_forall with different fourth expression.

Chapter 5 gives a brief overview of the entire work and concludes by giving remarks on different implementation and their performance in different architectures. Section 5.1 discusses the scope of extending this work.
2 Background

2.1 Partitioned Global Address Space Languages

Partitioned Global Address Space (PGAS) is a programming model which combines the advantages of shared memory model and message passing paradigm. All the languages developed with this model share many common concepts although they vary from one another in specific aspects like semantics and syntax.

PGAS languages assume that the number of threads is equal to the number of cores. Hence we will use the words threads and cores interchangeably throughout this session. Partitioned Global Address space languages, as the name suggests, partitions the shared address space in such a way that each core has a local portion of the shared address space. If each thread accesses only that portion of the shared memory in its memory space, then the program might actually be able to utilize the full potential of the architecture. The programmer controls data locality and is required to have explicit knowledge of the same. In addition to this each thread also has a private memory region which is accessible only by it.

The three main PGAS languages that are developed to run generically on machines from all vendors are, as mentioned in Chapter 1, Unified Parallel C, Co-Array Fortran and Titanium. For completeness we will discuss some of the features of Co-Array Fortran and Titanium before delving into Unified Parallel C.

Co-Array Fortran [12] is an extension to FORTRAN 95 which makes it as an efficient and robust parallel language. Converting a FORTRAN program into a Co-Array Fortran program is very simple. Each thread will have an image of the code, and the data associated with each of these images is different in different threads. To identify any of these images a special image index is available. This notion of image index is different from other PGAS languages and it could hinder the ease of programming to some extent, although programmers can manipulate the flow of the program in any of the image with the help of the image index. The image index can be considered as one more field which has to be considered while programming by the programmer.
Co-Array Fortran uses explicit synchronizations. A new object Co-array has been introduced in Co-array Fortran. Using this feature, data is distributed between different arrays. An element of an array from one image in a thread can be accessed by some other image from another thread by just specifying the thread number along with the array. All communications are single sided.

Titanium [13] is a PGAS language created by extending Java for a parallel environment. One of the main language features is that a global memory layout is specified by which programmer can control the processor affinity to data. Any two processes can read or write into each other’s memory directly. Loop iterations are not ordered and hence efficient optimizations could be made. Introduction of built in types for representation of many general domains, cross language support for many applications, deadlock prevention during compile time on barrier synchronizations and a wide variety of collective operations are some of the other features of Titanium.

Some of the common features of PGAS languages which are common to all the three languages are:

- They are parallel extensions of existing programming languages
- They support both shared and private variables. Shared variables are those accessible by all the threads and private variables are variables accessible by only those threads that declared them.
- Programmer has control of the data distribution in all PGAS languages
- Single sided communications, which are the most efficient form of communication for hardware that allows remote direct memory access (RDMA) are used in all the PGAS languages
- Distributed data structures are available.
- Common synchronization mechanisms are available to prevent deadlocks or race conditions
- Many of the widely used collective operations that are available in Message passing libraries are supported by these languages
2.2 Unified Parallel C

2.2.1 Overview

Unified parallel C (UPC) is an extension of ISO-C 99 [14] standard for parallel environments, which is based on the PGAS model. Of all the PGAS languages Unified Parallel C has received special attention in the past few years. Major vendors in the HPC field like HP, Cray and IBM already have their own commercial implementations of UPC. Many research institutions have also developed efficient implementations which are on a par with the commercial versions available. Berkeley’s UPC runtime and translator, MuPC [15], a runtime system provided by Michigan Technological University and GCC UPC [16] from Intrepid are among the other UPC distributions available. Research groups from University of Florida and George Washington University are also actively involved in the development of Unified Parallel C.

UPC was developed by extending C for parallel multiprocessor environment. It was developed from the experience obtained in developing three parallel programming languages Split C [17], AC [18] and PCP [19]. Unified parallel C is not the sum of all the features that were found in the above mentioned three languages. UPC is rather a parallel Programming language that was built by taking into consideration the best features of all these three languages.

The first specification for Unified Parallel C was introduced in 2001 [20] and it was implemented on a Cray T3E. The latest version available as of today is version 1.2 [21].

2.2.2 Language features

In this section we will look at the features of Unified Parallel C. Unified parallel C is an extended version of C and hence includes all the features of C. In addition to those some of the other features that are included in Unified Parallel C are discussed in this section.
2.2.2.1 SPMD model

UPC uses a Single Program Multiple Data (SPMD) model in which every thread has the same code but the flow of program and data present will vary from one thread to another. The flow of data can be controlled by specifying the thread which we intend to modify. This can be achieved by the using two special keywords.

The two special keywords that will almost be at use in any UPC program are MYTHREAD and THREADS. THREADS contain the value of the total number of threads used in the program. We can specify the number of threads statically at compile time or dynamically at runtime. If we are statically declaring the number of threads then THREADS could be used in the same way as, a constant is used in C. So arrays can have one or more of their dimensions as THREADS during declaration. On the other hand if number of threads is dynamically declared at runtime then THREADS can be used in only one of the dimensions during array declaration. MYTHREAD is a unique number which is used to identify each thread. It ranges from zero to THREADS - 1. The program will assume that there will be at least one thread which is mapped to a processing core available.

Each thread, in its memory space, has a private memory region which is accessible only by that thread. A shared memory region is also available which is accessible by all the threads. The common shared memory is divided into many partitions. Each of these partitions will reside in the memory space of a thread. So a thread is said to have affinity to that portion of the common shared space which resides in its memory. This is illustrated in the following figure.

![Figure 1 Memory layout in UPC](image-url)
Each thread, by using only that portion of the shared memory region which resides in its memory space increases the overall performance of the program. If a thread has to access a shared memory partition that does not reside in its memory space, the latency in accessing the remote shared memory is high, which results in loss of performance. This ability to utilize locality information is very important and ensuring data locality is the task of the programmer.

### 2.2.2.2 Variables, Arrays and Pointers

The private shared memory space in UPC is accessed the same way as variables would be accessed in C. In UPC the qualifier `shared` has to be used to specify that a variable should reside in the common shared memory region. All the scalars which are specified as shared are stored in the shared memory region of the thread with `MYTHREAD` value 0.

Arrays can also be declared as shared variables. When an array is declared as shared, its elements are distributed across the different processors in some order. The distribution can be controlled by the programmer depending on the needs of the program. The array is distributed in blocks. Block size is a compile time constant. We cannot specify the block size during runtime.

Various distributions can be specified while declaring arrays. These include blocked, block-cyclic, and cyclic. In a blocked distribution all the elements of an array are distributed in blocks such that every thread has affinity to a particular contiguous block of the array. Cyclic is the default distribution in UPC. If an array is declared shared without any block size specified then the default block size is assumed to be 1. That is the elements are distributed one by one across successive threads. If a block size is specified then the elements are divided as blocks and the blocks are distributed cyclically among all the threads. It is also possible to define an indefinitely blocked array which resides in thread 0. Some variable declarations in UPC have been explained below with the help of diagrams.

```plaintext
shared int A;  // Declaration of a shared scalar variable stored in thread 0
shared int arr[10];  // Declaration of shared array. Here block size is 1 by default
shared [3] int arr1[15];  // Declaration of shared array with block size 3
shared [] int arr2[5];  // Declaration of shared array with indefinite block size
```
shared int arr[10];

Figure 2  Cyclic distribution

shared [3] int arr1[15];

Figure 3  Blocked cyclic distribution
The pointer notation in UPC is slightly different from conventional C pointer usage. In UPC we have 4 types of pointers based on where the pointer resides and where it is pointing.

- A private pointer pointing to private memory space
- A private pointer pointing to shared memory space
- A shared pointer pointing to private memory space (which is logically wrong)
- A shared pointer pointing to a shared memory space

Examples of different types of pointer declaration in UPC are illustrated and explained below.

```c
int *ptr; // Private pointer pointing to private region. This is similar to C pointer syntax
shared int *ptr; // Private pointer pointing to shared region of memory
int *shared ptr; // Shared pointer pointing to a private space. Logically incorrect because of access violation
shared int *shared ptr; // shared pointer pointing to shared memory. The pointer resides in Thread 0;
```
Pointers which are pointing to shared objects are not like the traditional C pointers. They have additional fields to locate a shared object. The additional fields that has to be accommodated in a pointer to shared are

- The thread in which the object resides
- The virtual address of the block in that thread
- The phase which indicates the position of the element that the pointer is pointing to, in that block

There are special UPC functions available for obtaining specific information about shared pointer. With the help of these functions we would be able to obtain information related to shared pointer like, the thread to which the element pointing by the pointer has affinity to, the virtual address of the block in that thread where the pointer is pointing to, the phase value of the object in the block.

### 2.2.2.3 Work Sharing

UPC has a special for loop which is used for work sharing between different threads. This special for loop is `upc_forall()`. `upc_forall()` is a single valued collective operation. That is all the threads agree on how many iterations each thread should perform, which iterations each thread should perform and the sequence in which the iterations should be performed. It works the same way as a for loop in C, the only difference being the work is shared between all the threads available with the help of a fourth value in the loop. A for loop in C has 3 expressions: Initialization, condition and increment. UPC adds a 4 value to the traditional C loop which determines which thread has to work on which iteration. This fourth field is called the affinity field.

The fourth field of the `upc_forall()` could be any of the following

- An integer
- Pointer to shared type
- No value or continue keyword

When the affinity field is an integer say I, each thread executes those iterations where , I mod THREADS, equals MYTHREAD.

When the affinity field is a pointer to shared type say *shared ptr then each thread executes those iterations where `upc_threadof(ptr)` is equal to its thread number.
The function `upc_threadof()` takes as argument an address and returns the thread value where the address resides.

When the affinity field is `continue` or if no value is specified in the fourth field then all the threads execute all the iterations. In this case the `upc forall()` is not single valued collective operation. It is the same as a for loop.

### 2.2.2.4 Synchronisation

UPC provides various synchronisation constructs at various level to ensure that programs, with dependency between the various threads, are executed in the correct sequence. The various synchronisation constructs that are available in UPC are

- Barrier
- Notify & Wait
- Fence
- Lock and Unlock

**Barrier**

`upc_barrier` provides a synchronisation point where all the threads wait before proceeding to their next statement. It is a collective call and so should be executed by all the threads.

**Notify and Wait**

UPC supports split-phase barriers with `upc_notify` and `upc_wait`. First `upc_notify` is issued by all the threads and then an `upc_wait` is issued which completes the synchronisation. The threads may execute some statements in between the `upc_notify` and `upc_wait`. Immediately after `upc_notify` the next collective operation allowed is `upc_wait`. `upc_wait` can complete only after all the threads have executed `upc_notify`.

**Fence**

`upc_fence` ensures thread level synchronisation. It ensures that all shared memory accesses which were initiated before `upc_fence` call, are completed before `upc_fence` terminates.
Lock and Unlock

A lock is used to ensure that at a given instance only one thread has access to a shared element. upc_lock_t is the lock variable. With upc_lock() function a thread can acquire a lock and till it releases the lock, using upc_unlock(), all the shared variables it accesses, cannot be accessed by other threads.

2.2.2.5 Collectives

Collective operations are the most heavily used operations in many HPC applications. There are many collective operations supported in UPC few of which were discussed above. In this section we will discuss about all the collective operations in UPC.

The collective operations that are available in UPC are as follows:

- Broadcast
- Scatter
- Gather
- All Gather
- Exchange
- Permute
- Reduce
- Prefix reduce

Broadcast

upc_all_bcast() as the name suggests broadcasts a particular number of bytes from the shared memory region that has affinity to one thread to THREADS number of shared memory region each of which has affinity to one of the threads that take part in the collective operation.

Scatter

upc_all_scatter() divides a big block of contiguous shared memory region that has affinity to one thread into THREADS number of parts and sends it to different shared memory regions each of which has affinity to one of the threads that take part in the collective operation.
Gather

`upc_all_gather()` gathers `THREADS` number of small blocks of shared memory, each of which has affinity to one of the participating threads and copies it to a big block of contiguous shared memory region with affinity to a single thread.

All Gather

`upc_all_gather_all()` is very similar to `upc_all_gather`. The only difference is that a copy of the resulting big block is available in the `THREADS` number of shared memory region each of which has affinity to one of the participating threads.

Exchange

`upc_all_exchange()` copies $i^{th}$ block of memory with size `nbytes`, which has affinity to a thread $j$ into the $j^{th}$ block of memory that has affinity to $i^{th}$ thread.

Permute

We have an array, let’s say Perm, of size `THREADS` which should contain values ranging from zero to `THREADS` - 1 in any random order. When any array which is distributed across all the threads, is sent to `upc_all_permute()` function with the Perm array all the elements that has affinity to a particular thread say M, are copied into the thread whose value is stored in Perm[M]. All the entries in Perm array should be different.

Reduce

`upc_all_reduce()` is used to do some user defined operation like addition or multiplication across all the threads on a particular variable and store the resulting value in one of the threads.

Prefix reduce

`upc_all_prefix_reduce()` is same as `upc_all_reduce()` except that destination is an array of size `THREADS` and every element of the array will have its value updated. The value of an element in the destination array, say destination[M], is some operation done across all threads on a particular variable till thread M.

The last argument of the collective operations defines the synchronisation mode. Control of data synchronisation is done with the help of this argument. It can take in total nine values, which are obtained by performing an OR function between `UPC_IN_XSYNC` and `UPC_OUT_YSYNC`

where X and Y can take the following three values
• **NO**
• **MY**
• **ALL**

If X is specified as **NO** then the collective function can read/write when the first thread enters the collective operation. If **MY** is specified then the collective function can read/write only that part which has affinity to the threads that have entered the collective. When **ALL** is specified as the argument the collective function starts execution only when all threads enter the collective call.

If Y is specified as **NO** the collective function may read/write data till the last thread leaves the collective operation. If **MY** is specified then the collective function will return to the calling thread only after all read/write which have affinity to that thread is over. If **ALL** is specified the function call will return only after read/write associated with all the threads are completed.

### 2.2.2.6 Memory Allocation

Memory allocation, which is a heavily used operation in UPC, is handled in three different ways. We have all the memory allocation routines that C supports, to allocate private memory. Apart from that shared memory allocations are handled with specific memory allocation routines mentioned below.

The three memory allocation routines, to allocate shared memory, available in UPC are

- `upc_alloc()`
- `upc_global_alloc()`
- `upc_all_alloc()`

`upc_alloc()` allocates local portion of the shared memory. Any thread that makes a call to `upc_alloc()` allocates shared memory in its own memory region.

`upc_global_alloc()` is a non collective call for memory collection. The thread that invokes this function allocates shared memory for all the threads in their respective memory region.

`upc_all_alloc()` is a collective call for memory allocation. Every thread allocates its own portion of shared memory.
2.3 Benchmarking

Benchmarking can be defined as the act of running a computer program to analyse the performance, of a system, a specific component of a system or the program itself.

Benchmarks are very important because when we have to compare a computing resource across various platforms, it is not possible to do it with the information given in the specifications alone. This is true because often information on what we want to know might not be directly available to us. When we have to consider using high end computers it is very important that we use benchmarks to know beforehand the performance of the machines as the cost involved in using them are really high. So it is necessary that we develop a set of tests which could be run in different environments and could give us precisely the information that we are looking for.

Benchmarks are generally written to reproduce a certain type of workload, which the user might encounter, and obtain performance results based on that workload. In this way the user could exactly know the performance of his application in any system. Benchmarks are particularly important in designing and fine tuning hardware components like microprocessor, hard disks, interconnects and also the system as a whole. They are also important in developing efficient, application programs, programming languages, compilers and also in checking the performance of different implementations of the same specifications etc.

We will look, in the rest of this section, the challenges involved with benchmarking, and the different types of benchmarking available in general.

2.3.1 Challenges involved with Benchmarking

Benchmarking is not a trivial task as it involves deep understanding of what we intend to do. Interpretation of the results in particular is one of the difficult aspects of benchmarking. Many a time benchmarks that are written without considering all the aspects of the requirement often mislead the user than giving them any useful information. There are so many challenges involved with benchmarking. Some of them are discussed below.
• Certain benchmarks which are used to test the performance of hardware might not, in practice, give the actual performance of the hardware when it runs real time applications. For instance the LINPACK benchmark used to test supercomputers only gives us the theoretical peak value that could be achieved. We might not be able to get that performance when we run real time applications.
• Vendors may fine tune their product to give the best results for a particular benchmark.
• Benchmarks sometime focus only on raw processing speed and not on other important aspects like cost, power consumption, performance with respect to the problem at hand, I/O, memory size, reliability, scalability etc.
• Many vendor based benchmarks ignore important requirements like testing ability, disaster recovery and test only the kernel modules which are basic for production.
• A lot of benchmarks do not provide the information what the user is looking for. For example, let us consider a serial application which is to be parallelized into N sessions with each session running in a separate thread. If benchmarking is done individually upon these sessions user would be more interested in knowing the time taken for the slowest session. Giving the average time taken for all N sessions together would be of very little use to the user.

### 2.3.2 Different Types of Benchmarking

Benchmarks can be broadly classified into two types Synthetic benchmarks and Applications benchmarks.

#### 2.3.2.1 Synthetic Benchmarks

Synthetic benchmarks can be defined as benchmarks which measure the performance of a specific subsystem by isolating that subsystem from the rest of the system. In computing, a synthetic benchmark will generally look at the performance of a particular hardware like hard disk, main memory, processor, interconnect, GPU etc. or the performance of a particular component of the software or a combination of both hardware and software. The workload is designed in such a way that it utilizes only that component which is to be benchmarked, many number of times, to minimize the noise created by other components. Careful analysis should be done to remove all other factors, apart from what we intend to benchmark, which might influence the results.
2.3.2.2 Applications Benchmark

Application benchmarks are those in which real time, commonly used applications are run to evaluate the performance of a system. Many a times applications benchmarks often serve as a better yardstick than synthetic benchmarks in deciding the hardware, software or both, for real world scenarios. We can specifically test the system with a workload that will closely mimic the application which we intend to run.

2.4 Existing Benchmarks

In this section we will look at some of the existing benchmarks that are already used to test the performance of UPC.

2.4.1 GWU Benchmarks

The first step in benchmarking UPC for performance analysis was done by El-Ghazawi and Chauvin [22]. Their work describes an early version of a benchmark UPC_Bench which was designed to analyse the UPC compilers performance. UPC_Bench is a combination of synthetic benchmarks (UPC_Synthetic) and applications benchmarks (UPC_Applications). The synthetic benchmark focuses on benchmarking data access in UPC. UPC_Synthetic uses some concepts of STREAM benchmarks to test for performance, the access of a private data, access of local shared data which resides in a thread’s own memory space and access of remote shared data which is shared data that resides in some other thread’s memory space.

The Applications benchmark (UPC_Applications) tests three parallel applications

- N queens Problem, which is an embarrassingly parallel application which has very less remote memory access
- Sobel edge detection problem, which involves a reasonable amount of remote memory access (access of halo region), and
- Matrix multiplications which involves heavy remote access of data.
Their work helped in understanding a major problem with HP UPC compilers which was that the time take to access local shared memory and remote shared memory was the same. This work also covered some compiler optimizations like remote reference pre-fetching and aggregation.

El-Ghazawi and Contonnet [23] discuss the various performance issues in UPC with the help of NPB. Various NAS parallel benchmarks were implemented with different workloads. A performance comparison is made with NPB 2.3 MPI implementation.

With the help of STREAM, GUPS and NAS Parallel benchmarks, Contonnet and El-Ghazawi discuss [24] the performance of UPC with and without, some proposed improvements for memory translation. Their work also included comparison of UPC with some MPI implementations. “CG, EP, MG, FT, and IS” were the five main kernels from the NAS Parallel Benchmarks [25] which were rewritten in UPC and tested.

### 2.4.2 Berkeley Benchmarks

University Of California Berkeley is an active participant in the development of Unified Parallel C. They have developed a runtime and a translator which runs in many of the modern architectures including Cray XT series, IBM Bluegene series, AIX SMP’s, SGI Altix, Cray X1 etc.

They have developed some benchmarks to evaluate the performance of their compiler. In this section we will look into some of their work.

A performance analysis of the BUPC compiler [26] was made and it was compared against the performance of the HP compiler. In this work they have presented a set of micro-benchmarks, synthetic and application benchmarks.

#### 2.4.2.1 Micro-benchmark

The micro-benchmark was used to find out the overhead associated with some of their programming constructs. The three main things that were tested are

- UPC’s special for loop
- Dynamic allocation
- Pointer to shared operations
As already discussed UPC has a special for loop `upc_forall` which is a collective operation. `upc_forall`, being one of the heavily used constructs in UPC, was tested for performance results. It was tested with four different fourth expression values. The four different values are

- integer fourth expression
- without any value for the fourth expression
- address of a scalar shared variable
- address of different elements of a shared array

Dynamic allocation of shared memory is also an operation that could be seen widely in many UPC codes. As discussed above the three different constructs available to allocate memory are

- `upc_alloc`
- `upc_global_alloc`
- `upc_all_alloc`

All these three functions are benchmarked and performance of the collective `upc_all_alloc` has been given special attention as it is most expensive of all the three.

Pointer to shared operations is by far the most important functionality in UPC as its programming style encourages pointer dereferencing to fetch objects in remote shared memory or local shared memory region. So it is benchmarked to find out the overhead associated with it. Also some important optimizations to the pointer to shared operations are done which improves the performance of the remote shared access.

### 2.4.2.2 Synthetic and Application Benchmarks

A set of synthetic and application benchmarks were also implemented in BUPC. Some of them that were implemented and tested are

- `vector add`
- `gups pair`
- `scale`
- `ep`
- `is`
- `npbcg`
- `cg`
- `mg`
In vector add 2 vectors (arrays) are added and the result is stored in a third vector. The performance of local shared memory access is assessed with this benchmark as the vectors are aligned to avoid communication.

**Giga-updates per second (gups)** pair was used to test the performance of remote shared address access. Randomly selected array indexes in which the percentage of remote shared access could be adjusted were used for this benchmark. Two reads and two floating point operations were done on a remote shared memory.

**scale** is similar to gups. It varies only in that; it also updates the shared variable in addition to what gups does. So an extra write takes place every time in addition to a read and a multiplication.

**ep** stands for embarrassingly parallel. This was developed based on the NAS EP benchmark. This involves very little communication and is basically to test the scalability of UPC.

**npbcg** and **cg** are conjugate gradients with different communication pattern. npbcg is conjugate gradient with bulk writes. cg is conjugate gradient with small reads and some writes.

**mg** is UPC implementation of MG benchmark from the NAS. Communication consists of mainly Halo exchanges in this benchmark. In both **mg** and **npbcg** upc_memput of large messages are used.

**is** is the integer sort benchmark from NAS parallel benchmarks where communication happens by the use of upc_memget bulk memory transfer routine.

### 2.5 Motivation

The PGAS languages, especially UPC is gaining popularity and is seriously considered to be the programming language for the next generation high end computers.

It is important that we understand the strengths and weakness of the different implementations available. Although applications benchmark provide a rough idea of which implementation performs better for which application, it is only with the help of synthetic or micro-benchmarks, it is really possible to find out exactly which part of the implementation has to be fine tuned. Not much work has been done in the past to test the overhead associated with each of the function/operations of UPC. This benchmark suite is developed to address this issue.
3 Design & Implementation

The objective of this micro-benchmark is to effectively find out the overheads associated with some of the commonly used programming constructs in UPC. We try to eliminate all the other timing overheads in our code except for that of the construct in test. We will discuss throughout this session of how we can do it for each of the construct considered. In some cases other approaches apart from what has been implemented is also suggested.

Many of the ideas used this benchmark are based on the implementation of the EPCC’s micro-benchmark suite for OpenMP [27].

This chapter is structured in the following order. Section 3.1 discusses about the design of the timing routine used throughout the benchmark. From section 3.2 to 3.4 we discuss the design of the benchmarks for the various programming constructs which we have implemented in this suite. Section 3.5 gives a brief overview about all the implementation details.

3.1 Design of Timing Routine

In our benchmark we have a generic timing routine, which is used by all the other functions, to get a reference time. We find out the time taken by a particular section of the code by making a call to the timing routine before the start of the section and after the end of the section. The difference between the latter and the former values give the time taken for the execution of that particular section.

A pseudo code given below shows how this is done
This pseudo code measures the time taken for the code section to execute N number of times including the overhead associated with the for loop.

The timing routine uses C inbuilt function `gettimeofday()` to get the current time, expressed in seconds and microseconds. It then converts the values obtained into a double value, by adding the microseconds and seconds, and returns the double value to the calling function. During the first call to this routine the double value obtained is stored as a static variable. This static variable represents the initial time. All calls from that point, during the execution of the program, to the timing routine will return a double value which is the difference between the current time obtained and the initial time.

### 3.2 Upc_forall

As discussed earlier `upc_forall` is the special for loop available in UPC with an additional fourth expression. This fourth expression determines which iteration has to run in which thread. The fourth expression as discussed in previous sections can take an integer value, no value or address of, shared scalar or array. The benchmark tests for the fourth expression with an integer value, no value and address of successive elements of a shared array which has the following distribution.

```c
shared int a[M];
```
Each thread, from zero to THREADS - 1, has M/THREADS elements of the array a, in its
local shared memory region, which is distributed in cyclic order. M is a fixed value that
could be changed.

We are interested in finding out the time associated with the execution of the
upc_forall loop alone. There are two approaches of doing this. They are described in
Sections 3.2.1 and 3.2.2.

3.2.1 Use of a dummy routine

A dummy routine is created which runs for some considerable amount of time. This
dummy routine is then placed inside an ordinary for loop and run for N/THREADS
iterations in a single thread. The time taken to run the N/THREADS iterations is noted.
The dummy routine is then placed in upc_forall and run for N number of iterations
so that each thread runs N/THREADS iterations.

The time difference between the time taken for upc_forall with dummy routine in a
thread, and the time taken for the dummy routine in a for loop in that same thread gives
us an approximate value of the time taken by the upc_forall loop. Care has to be
taken as the overhead associated with the ordinary C for loop is also added here. We
assume that the overhead due to the for loop is negligible. The overhead is calculated in
the following way

\[ T_{FA} = T_{Nfa} - T_{NT} \]

Where,

\[ T_{FA} \quad = \quad \text{Overhead associated with one iteration of the upc_forall loop} \]
\[ T_{Nfa} \quad = \quad \text{Time taken by N iterations of upc_forall with dummy routine} \]
\[ T_{NT} \quad = \quad \text{Time taken by N/THREADS iterations of the for loop with dummy routine} \]

3.2.2 Use of single variable update

Another approach to the problem could be the use of just a variable update inside the
upc_forall. Instead of having a reference time (time taken by the dummy routine in a
single thread for N iterations) in this approach we directly find the absolute time taken by
the upc forall loop. The variable update is to prevent the C compiler from throwing away the for loop. We should remember that we are trying to assess the time associated with running the loop N number of times across THREADS number of threads where only N/THREADS iterations are run in a single thread. The overhead associated could be calculated in the following manner,

\[ T_{FA} = \frac{T_N}{N} \]

Where,

\( T_{FA} \) = Overhead associated with one iteration of the upc forall loop

\( T_N \) = Time taken to execute N iterations of the upc forall with a variable update

N = Total number of iterations

The second approach “Use of a single variable update” has been implemented in the benchmark although both the approaches were tested. This is so because the first approach is not as robust as the second approach for finding the time associated with upc forall. When calculating the time difference between the upc forall and the native for loop, as the for loop was taking more time than the upc forall, we obtained negative results for the time taken by upc forall.

### 3.3 Synchronisation Constructs

UPC provides various synchronisation constructs, which are essential for programs with data dependency. The four constructs that the benchmark addresses are

- Barrier
- Notify and Wait
- Lock and Unlock

Our objective is to find out the overhead associated with each of these constructs.
3.3.1 Barrier

As mentioned in the previous section a barrier can be thought of as a synchronisation point where all threads wait before proceeding to the next statement. A barrier could be benchmarked by simply executing it $N$ number of times, placing it inside a for loop. In this way every time the loop is run all the threads are made to wait at the barrier. The time taken by for loop is negligible when compared to the time taken by the barrier. The time taken to run a barrier can be calculated in the following way

$$T_B = \frac{T_N}{N}$$

Where,

$T_B = \text{Overhead associated with a single barrier statement}$

$T_N = \text{Time taken to execute N times, the barrier statement}$

$N = \text{Total number of iterations}$

3.3.2 Notify and Wait

Notify and wait are the split phase barriers available in UPC. They do the same functionality as a barrier except for that in between a Notify and Wait, certain computations could be placed. The specification specifies that there cannot be any collective operations in between a Notify and Wait pair. The overhead associated with executing, `upc_notify` and `upc_wait`, can be calculated in a similar way like that of an `upc_barrier`. We have to execute $N$ number of times, `upc_notify` and `upc_wait` pair. The time taken to execute them is divided by $N$ to obtain the overhead associated with running the `upc_notify` and `upc_wait` once.

$$T_{NW} = \frac{T_N}{N}$$
Where,

\[ T_{NW} = \text{Overhead associated with Notify and wait statements} \]

\[ T_N = \text{Time taken to execute N times, the upc\_wait and upc\_notify pair} \]

\[ N = \text{Total number of iterations} \]

### 3.3.3 Lock and Unlock

As discussed earlier UPC provides support for lock and unlock routines. Any shared variable that is being accessed by a thread, after acquiring a lock, will not be available for other threads to read or to modify till this thread releases the lock.

We would like to test the overhead associated with the acquiring and releasing a lock. The following method could be used to do it.

A dummy routine which takes a finite time to execute is run in serial some N number of times. Then the same dummy routine is run in parallel in each thread \( N/\text{THREADS} \) number of times. The dummy routine is placed within a lock and an unlock routine. We calculate the timing involved in executing the code in parallel by waiting till all the threads acquire and release the lock. Then we calculate the difference between the time taken by the parallel code and time taken by the serial code which is the overhead associated with the Lock and Unlock routines. The overhead can be calculated in the following way

\[ T_{Lck} = T_{Par} - T_{ser} \]

Where,

\[ T_{Lck} = \text{Overhead associated with acquiring and releasing a lock} \]

\[ T_{Par} = \text{Time taken to run the code in parallel with locks. Each thread executes the code } N/\text{THREADS} \text{ times} \]

\[ T_{ser} = \text{Time taken to execute the code in serial N number of times} \]
3.4 Collective Operations

In our benchmark we have considered seven collective operations for which the overheads associated are calculated. All the collectives are benchmarked in the same way. The difference is only in the data types each collective takes as arguments. More about that is discussed in the implementation section. We present a general design for all seven collectives here.

As mentioned earlier the value that the collective operations take as their last argument determines the synchronisation mode in UPC. The collective operations could be executed with no synchronisation, synchronisation between all the threads or synchronisation at thread level.

The collective operation to be benchmarked is executed $N$ number of times and the total time taken to execute the operation is noted. If there are any other operations (operations like barrier or, notify and wait, etc.), whose time overhead were also measured along with the collective operation, the overhead associated with them are subtracted from the total time taken. We will refer to these other operations as OTH. The overhead associated with OTH are calculated by executing them, without the collective, $N$ number of times.

\[
T_{coll} = \frac{(T_{tot} - T_{oth})}{N}
\]

$T_{coll} = \text{Overhead associated with executing the collective}$

$T_{tot} = \text{Total time taken to execute, collective and OTH, } N \text{ times}$

$T_{oth} = \text{Time taken to execute OTH, } N \text{ times}$

3.5 Implementation details

The micro-benchmark has been designed, as seen in the previous section, in a simple but effective way. In this section we will look into how the design was implemented in this benchmark. This section covers in depth, the data structures used, data sizes which were tested and all other implementation details.
The benchmark is split into three executables. Each executable covers all the operations that belong to a particular class of programming constructs or operations. The three classes of constructs/operations are

- Collective operations
- Synchronisation constructs
- Work sharing construct

All the optimizations that the compiler is capable of doing have been turned off by the usage of pre-processor directives and compiler flags.

All the routines have been tested for correctness during the development stage of the benchmark. Exhaustive testing has been done to

- Test the correctness of the output of all the collective operations
- Test the affinity of the blocks of arrays to the corresponding threads
- To test that the file operations involved are run correctly

### 3.5.1 Usage of benchmark

There are subroutines associated with each collective. The execution of these subroutines is controlled arbitrarily by a file that is being called from the collective executable. The file that needs to be modified in order to define which subroutine is to be run is collfile.sh. This file contains numerical values only. The first number in collfile.sh is the total number of subroutines to run. From the second number onwards, each number corresponds to a particular subroutine. By specifying the numbers corresponding to subroutines, the subroutines are run in the specified order. The mapping between the numbers and subroutines is as follows:
<table>
<thead>
<tr>
<th></th>
<th>Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Scatter</td>
</tr>
<tr>
<td>3</td>
<td>Gather</td>
</tr>
<tr>
<td>4</td>
<td>All Gather</td>
</tr>
<tr>
<td>5</td>
<td>Exchange</td>
</tr>
<tr>
<td>6</td>
<td>Permute</td>
</tr>
<tr>
<td>7</td>
<td>Reduce</td>
</tr>
</tbody>
</table>

**Table 1  Numbers corresponding to the subroutines for collectives**

Similarly the sub routines in the forall executable can be controlled by modifying the forall_file.sh.

**Compilation steps**

A Makefile is available which can be edited in the following way to compile programs.

**Step 1** Assign to the SRC field of the Makefile the source file’s name

**Step 2** Assign to the EXE field of the Makefile the name for the compilation target

**Step 3** Assign to the CC field the compiler to be used. *upcc* if the compiler is Berkeley UPC and *cc –h upc* if the compiler is Cray UPC

**Step 4** If Cray UPC is to be used the default program environment should be changed from PGI to Cray using *module switch* command

**Step 5** If Cray X2 has to be used then *module add* command should be used to change the environment from XT to X2
3.5.2 Data Size & Timing

The benchmark tests all the collective operations for different data size. Every collective operation for a particular processor count is designed to test data from “4*2^0” to “4*2^k” where “k” is any positive integer. This range can be easily altered by changing the value of “SIZE” which is defined during the start of the program. As we are using integers in collective operations the message size is a multiple of 4 bytes.

The number of iterations for which the benchmark is run is determined dynamically on the fly. The benchmark has been implemented in the following way.

- We initially set the number of iterations to a small number 32 or 64 in this benchmark
- We check if the time taken by the section of code, which we intend to benchmark, is not less than 0.1 seconds for the given number of iterations.
- If the time taken is less than 0.1 seconds we multiply the iteration count by 2 and again run that particular section of the benchmark.

3.5.3 Data Structures

The benchmark design requires dynamic allocation of shared arrays with variable block size during compile time. In UPC, there is no straight forward way available to dynamically allocate an array with block size given during compile time. Using the memory allocation routines it is possible to create arrays with the default block-size layout, which is cyclic. Every thread, with MYTHREAD value 0 to THREADS - 1, receives the successive element of the array. But for the collective operations we want the arrays to have contiguous elements of an array. To contiguously allocate memory in each thread the following data structure was used.
In Figure 6, B is any variable name that we declare. We allocate memory to B using upc_global_alloc or upc_all_alloc in the following way.

\[
B = \text{upc\_global\_alloc}(\text{THREADS}, \text{mess\_size}\times\text{sizeof(int)})
\]

\[
B = \text{upc\_all\_alloc}(\text{THREADS}, \text{mess\_size}\times\text{sizeof(int)})
\]

This allocates, to each thread, a structure containing mess_size integers packed contiguously. We use array of structures because memory for structures is allocated contiguously on a given thread, unlike arrays which are allocated with cyclic distribution. Since for the array elements the default distribution is cyclic, the first dimension is distributed across all the threads and the blocks of second dimension are allocated contiguously in each thread.

Now B has the same memory layout as the static array

\[
\text{shared [mess\_size] int } B[\text{THREADS}][\text{mess\_size}];
\]

It is essential for collective operations that the memory layout is in the following way as the results are based on the memory layout.
4 Performance Analysis

In this section we discuss the results of our benchmark. The benchmark was run in two different architectures with two different compilers. The XT4 and X2 parts of Hector, UK’s National supercomputer were used to carry on this project. Cray’s UPC compiler and Berkeley’s UPC compiler were used in the XT4 part and, in the X2 part Cray’s compiler was used. The results of all the experiments are discussed below. The benchmark has been tested for the following processor counts: 4, 8, 16, 32, 64, 128, 256 on XT and 4, 8, 16, 32, 64, 112 on X2.

This chapter is structured in the following way. Section 4.1 gives a brief overview of the two architectures in which the micro-benchmark was run. Section 4.2 discusses about the performance of collective operations in BUPC and Cray compilers. In Section 4.3 the performance of the synchronisation constructs of both the compilers in the two architectures are discussed. In Section 4.4 performance of the special for loop upc_forall is discussed.

4.1 Architecture overview

The two architectures that were used for this project are Cray XT4 and Cray X2. A brief description about both the architectures is given below. Hector uses Unicos/lc operating system which consists of Compute node Linux and a fully featured Linux distribution. The compute node Linux is the light weight kernel which is used in back end nodes. The Linux distribution is used in the service nodes available which include the IO nodes and system nodes.

4.1.1 Cray XT4

Cray XT4 uses quad core AMD Opterons each of which has a clock rate of 2.3 GHz. The theoretical peak performance achievable by each of these quad core processors is 36.8
gigaflops per processor for double precision numbers. The Level1 cache and Level2 cache are private caches whereas Level3 cache is shared by all the four cores in the processor. There are a total of 5664 AMD quad cores present in the XT4 system and hence total of 22,656 cores of available out of which a maximum of 16384 can be used in one run. The theoretical peak performance offered by Hector is 208 teraflops.

Each quad core processor gets 8 GB of main memory and hence each core gets 2 GB of main memory. All the processors are connected using Cray seastar2 communication chips. Every quad core processor has its own seastar2 interconnect chip. Hector has 12 I/O nodes. Lustre distributed parallel file system is used to access disks.

4.1.2 Cray X2

Hector includes a X2 processing component known as “Blackwidow”. This has 112 vector processing cores split over 28 nodes each having 4 processing units. The vector processing units have a clock frequency of 1.6 GHz. A processing unit will have in it 8 vector pipes, a 4 way dispatch superscalar core, and Level1 and Level2 caches. One node has one Level3 cache which is shared by all the processing elements of the nodes. The superscalar cache has access to 16 KB data and 16 KB instruction cache. The interconnection network of Cray X2 is a radix-64 folded–Clos (fat tree) low latency network which allows each processor to access memory anywhere in the system. Each X2 node has access to 30 GB of memory so each processing element in it has access to 7.5 GB of memory.

4.2 Collectives

We will first discuss about the performance of collective operations. The collectives were run on both, BUPC and Cray compilers. The Cray compiler in XT4 does not support collective operations. So data collection from XT4 was with Berkeley UPC compiler. In the X2 only Cray compiler is available. So data collection from the X2 was with the Cray UPC compiler. All the graphs presented have their Y axis in logarithmic scale. Therefore care should be taken while interpreting the results. The message size given is in words and not in bytes. Therefore message size should be multiplied by 4 to get the number of bytes information.
Figure 7  Performance of Broadcast on Cray XT4 (Berkeley implementation)

Figure 8  Performance of Broadcast on Cray X2 (Cray implementation)
4.2.1 Broadcast

In Figure 7 we can see that time taken for the upc_all_broadcast() operation almost remains the same till a certain point, undergoes a sudden transition and starts increasing exponentially after that. We will refer to this region, where there is a sudden transition in timing as the “transition region” throughout this chapter. The sudden increase in timing might be because of some change in protocol in the underlying layer.

The increase in time after a particular point will be referred as the “threshold effect” from here onwards. This is because the message size does not affect the timing results till a particular threshold.

The overhead associated due to other factors is much more pronounced than the overhead associated with increase in message size which is negligible. After the threshold is reached, the message size starts effectively contributing to the time taken by the collective operation. Doubling of message size, from the point where threshold is reached, causes the timing to almost double. In Figure 7 we can see that for 4 processors the transition region lies between message sizes of 4096 and 8192 words.
The slope of the transition region is high for lower number of processor count. The slope decreases as the processor count increases.

When the number of processors is doubled the transition region shifts towards smaller message sizes. We can also notice that in every case the product of the processor count and the message size at which the transition effect starts is a constant.

In Figure 8 we see that timing is constant initially and it starts increasing exponentially after a particular threshold. Unlike the BUPC implementation in XT we don’t see a transition region with high slope in the graph.

From Figure 9 we can observe that for a fixed message size the Cray compiler in the vector machine performs better than the Berkeley UPC in XT4 for the same number of processor count. This may be because of the difference in architectures. The Cray collective is almost better than BUPC by a factor of 10. We should however understand that we are comparing two different architectures and so a direct comparison between timing may not be useful. Comparing the scalability in both the architectures may be a better way to compare the results. Both the compilers seem to scale well with the increase in processors.

### 4.2.2 Scatter

We observe that the performance of the `upc_all_scatter()` operation is almost the same as the `upc_all_broadcast()` operation. The timing associated with the `upc_all_scatter()` operation is very similar to the timing of the `upc_all_broadcast()` operation for both the compilers in the two architectures.

The scatter operation takes slightly more time than the broadcast operation. This could be comprehended easily because in broadcast there is no other operation involved except for copying but in scatter there are a few other operations involved apart from copying. This includes splitting of the array which resides in the thread with `MYTHREAD` value 0 to `THREADS` number of parts. This is not a big overhead and so the performance of scatter does not vary much from broadcast. In Figures, 10 and 11, we see the threshold effect that we saw in Figures, 7 and 8. We can see in figure 10 that there is a transition region with higher slope value for smaller processor count which decreases as the processor counts increase.

We see from Figure 12 that the scalability of the scatter operation for a fixed message size in both BUPC and Cray implementation is almost the same, with the Cray implementation taking less time in the vector machines as in the broadcast operation.
Figure 10  Performance of Scatter on Cray XT4 (Berkeley implementation)

Figure 11  Performance of Scatter on Cray X2 (Cray implementation)
Figure 12    Time vs. Processors for Scatter operation (16384 words)

Figure 13    Performance of Gather on Cray XT4 (Berkeley implementation)
Figure 14  Performance of Gather on Cray X2 (Cray implementation)

Figure 15  Time vs. Processors for Gather operation (16384 words)
4.2.3 Gather

The performance of `upc_all_gather()` is, as expected, similar to the performance of the `upc_all_scatter()` operation. Gather being a very similar operation in terms of implementation does not show any odd behaviour. The implementation details might not vary a lot for scatter and gather. So the performance of gather is identical to that of a scatter operation in both the implementations for both the architectures.

We can infer from figure 15 that the scalability of gather is also the same as that of scatter. The gather operation in X2 with the Cray implementation takes less time than the BUPC implementation of the gather operation in the XT4.

4.2.4 All Gather

An allgather operation has `THREADS` time more copying to do as all the threads get a copy of the result, of the gather operation. So it is obviously an expensive operation and the overhead associated with it will be more than all the operations discussed before.

In all the operations that we have seen previously the vector processors performed better than the scalar processors. Figure 18 shows that the BUPC implementation on XT4 scales better when compared to the Cray implementation on X2. The Cray implementation in the vector processor takes almost the same time as that of the BUPC implementation in scalar processors.

Another important aspect to note is that in all the collective operations discussed above, as the number of processors increase the slope of the transition region decreases. If we look at Figures 7, 10 and 13 we can see that for a processor count of 256 the graph was much smoother than for a processor count of 4. However in Figure 16 we notice that as the processor counts increase, slope of the transition region also increases as opposed to the behaviour of all the collectives discussed above.

As noted earlier the sudden increase in timing could be because of some protocol change, probably from PUSH to PULL implementation or vice versa. In a PUSH implementation all the processors involved in the collective put their values to the processor in which the final result is stored and in PULL implementation the processor which contains the final output gets from all the processors the value which they are expected to contribute.
Figure 16  Performance of Allgather on Cray XT4 (Berkeley implementation)

Figure 17  Performance of Allgather on Cray X2 (Cray implementation)
Figure 18  Time vs. Processor for Allgather operation (16384 words)

Figure 19  Performance of Exchange on Cray XT4 (Berkeley implementation)
Figure 20  Performance of Exchange on Cray X2 (Cray Implementation)

Figure 21  Time vs. Processors for Exchange operation (512 words)
4.2.5 Exchange

Exchange is one of the most widely used collective operations and also one of the most time consuming operations. It is the UPC equivalent of All to All operation. It is important to find out the overhead associated with this construct.

Looking at Figure 19 and 20 we can observe that performance of exchange almost follows the same behaviour as the allgather operation. We see that the slope of the transition region increases with increase in message size for the BUPC implementation. Also the threshold effect in X2 is same as in allgather operation.

This similarity in performance of upc_all_exchange and upc_all_gather_all is reasonable as the messages size of exchange is almost similar to allgather. In both BUPC and Cray implementation as the message size increases the curve rises exponentially, in a log scale. So when we use exchange care should be taken if the message size is high.

4.2.6 Permute

It is quite interesting to note the performance of permute operation in XT4 and X2. Figure 24 shows the Cray implementation of permute, where we see an exponential increase in the time taken for permute operation after a particular threshold.

On the other hand in BUPC implementation (Figure 23) we see that the time taken almost remains constant except for lower processors counts, where the timing increases at around 8192 words.

We can observe from Figure 22 that for higher message size the time taken by the X2 is more than the time taken by the XT4 for message size of 16384 words. Of all the collectives that we have seen till now this is the only operation where the performance of the X2 is worse than the performance of XT4. So we can conclude that BUPC is implemented in a better way than the Cray UPC for permute operations.

Although the performance of the permute in Cray implementation is same as that of broadcast or scatter in X2, the performance of permute in BUPC is much better than the performance of broadcast or scatter in XT4.
Figure 22  Performance of Permute on Cray XT4 (Berkeley implementation)

Figure 23  Performance of Permute on Cray X2 (Cray implementation)
Figure 24  Time vs. Processors for Permute operation (16384 words)

Figure 25  Performance of Reduce on XT4 (Berkeley implementation)
Figure 26  Performance of Reduce on X2 (Cray implementation)

Figure 27  Time vs. Processors for Reduce operation (16384 words)
4.2.7 Reduce

One of the most interesting analysis is with the reduce function. The Berkeley compiler seems to have implemented it very effectively. We see from Figure 25 that the time taken by the collective is almost the same for 1 word and 16384 words for any processor count.

In Figure 26 we observe the threshold effect after a particular message size for all the processor counts.

The Cray implementation on the other hand scales very well as compared to the Berkeley implementation. From Figure 27 we can observe that the Cray implementation takes almost the same time for all the processor count whereas the time taken by the Berkeley implementation is linear as the processor counts increase. The time taken in X2 for all processor count is almost equal to the time taken by the BUPC implementation in 256 processors. Hence BUPC implementation is still better for the given dataset.

4.3 Synchronisation Constructs

The three synchronisation constructs have been benchmarked in X2 and XT4. In XT4 both the Cray and BUPC compilers were used whereas in X2 only the Cray compiler was used.

4.3.1 Lock and Unlock

The implementation of locks is pretty good in the Berkeley implementation. The time taken remains almost a constant in the BUPC implementation. The Cray implementation seems to scale really bad. With increase in number of processors the value of the time taken to execute locks increases exponentially in both, vector and scalar processors. In Figure 28 we can see the threshold effect taking place when the number of processors used is more than 32.

We should, however, note that the time taken to execute locks in X2 with the Cray implementation is much less than the BUPC implementation in XT4. Considering
scalability and the performance in the scalar processors we can conclude that the BUPC implementation of locks is much better than the Cray implementation.

![Figure 28](image)  
**Figure 28**  
Lock and Unlock on Cray X2 and XT4

### 4.3.2 Barrier

We see in Figure 29 that the time taken for the barrier is almost the same in both the compilers. We can also see that the time take for executing a barrier in the vector processor is much cheaper than executing it in scalar processors. The time taken is exponential in scalar and the vector processors. So in terms of performance for barrier implementation we can say that both Cray and BUPC are at the same level.
Figure 29  Barrier on Cray X2 and XT4

Figure 30  Notify and Wait on Cray XT4 and X2
4.3.3 Notify and Wait

A wait and notify is expected to take the same time as that of the barrier. We also see from Figure 30 that this is indeed the case in both the architectures for the 2 implementations. The behaviour of the wait and notify synchronisation construct is as expected the same as barrier.

4.4 Upc forall

The time taken by the upc forall loop is noted down in the available two compilers for the two different architectures.

<table>
<thead>
<tr>
<th>Fourth value</th>
<th>Cray XT</th>
<th>BUPC XT</th>
<th>Cray X2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer expression</td>
<td>3.83 ns</td>
<td>42.1 ns</td>
<td>94.4 ns</td>
</tr>
<tr>
<td>No value</td>
<td>3.22 ns</td>
<td>1.94 ns</td>
<td>81.1 ns</td>
</tr>
<tr>
<td>Shared address</td>
<td>56.6 ns</td>
<td>60.3 ns</td>
<td>221 ns</td>
</tr>
</tbody>
</table>

Table 2 Timing for upc forall in the two compilers in nanoseconds

The upc forall loop with different fourth expression has been presented here. The time taken in the X2 compiler is large for all three values of the fourth expression. In all three cases the time taken by the shared address is more than the other two values. The BUPC implementation is the most interesting one as the interval between the timing, of having an integer expression and not having a fourth value is most predominant in it. The Cray compiler seems to perform better than the BUPC compiler in the XT4. In X2 the Cray compiler’s performance is not good in comparison with its performance in XT4.

In Table 2 the performance of the upc forall without a fourth value is same as the performance of the for loop of the underlying C compiler. We can conclude that PGI implementation of for loop is better than the Cray implementation and also Cray’s implementation of for loop for the vector processor is bad.
5 Conclusion

MPI, the most widely used specification for programming supercomputers as of today has many shortcomings with respect to scalability which has made researchers look for better alternatives for the exaflop era. A new programming model called Partitioned Global Address Space model was developed to address the issue.

UPC, one of the widely known PGAS languages, which is still in its research stages, is being looked upon by many researchers as the future programming language for supercomputers with thousand of cores in them.

This project was undertaken to test the performance of the different UPC compilers available and analyse the results to suggest the areas of improvements in those compilers.

The micro-benchmark suite has been designed and implemented to test the overhead associated with various programming constructs of UPC. A general comparison is made with the available two compilers, Berkeley UPC and Cray UPC. After analysing the results from both the compilers in two different architectures the following results have been drawn. In the implementation of collective operations both the compilers seem to have the same performance with BUPC performing better for “permute” and “reduce” operations.

The BUPC implementation of locks seems to be much better than Cray’s implementation which scales poorly as the processor count increases. The performance of both the compilers is identical as far as barrier, notify and wait operations are concerned.

The performance of the special for loop upc_forall is better in the Cray implementation of UPC for the XT4 machine.

Of the three cases, Cray implementation on XT4, Cray implementation on X2 and Berkeley implementation on XT, we saw that the performance of the Cray implementation on X2 was the best for collectives and also for the synchronisation constructs.

Cray’s superior performance in X2 is mainly because of the architecture of the machine. Non uniform access of memory with other vector processors in the high bandwidth low latency interconnection network, which allows any processor to access global memory in a fast manner, and cache coherency across all the X2 blades could be the reason for
superior performance of UPC. Support for UPC is in both hardware and software level in Cray X2.

Currently the Berkeley implementation, which can be run in any platform, is better than the Cray’s implementation in the XT4 as BUPC’s performance in the Cray machine is similar to or in some instances better than the Cray’s implementation which is specifically designed for the Cray machines. The performance of locks is especially much better than Cray’s implementation.

As of today Cray UPC is supported only in software but not in hardware level on the XT. The Seastar in Cray XT4 is basically an MPI engine, designed for supporting non-blocking MPI. It does not support single sided communications in hardware and since UPC uses single sided communication, Cray could not come up with an implementation that uses the hardware capabilities of the machine.

To address the issue of support for PGAS languages, Cray has come with the next generation of interconnects, Gemini. Gemini supports AMOs (Active Memory Operations), by which some select operations can be sent to the data’s home memory controller and executed there. By minimizing inter-node and intra-node memory traffic and eliminating significant number of coherent messages various opportunities for parallelism are created. This support for RDMA operations is best suited for single sided communications which are used by PGAS languages and this might increase the productivity of Cray implementation of UPC.

5.1 Future work

There is a lot of potential in benchmarking UPC as not much has been done in this area. This work addressed the performances of UPC in Cray XT4 and X2 systems. The future work that is possible in benchmarking UPC with the help of this micro-benchmark suite and also proposed extensions to the micro-benchmark suite itself is given below.

The benchmark has covered testing most of the functions/operations of UPC. The benchmark, however, hasn’t covered the memory allocation routines. It would be interesting to develop a benchmark that could find out the time associated with allocating memory. A comparison between upc_all_alloc() and upc_global_alloc() should be particularly interesting as research is going on to increase the performance of the one-sided collective operations like upc_global_alloc(). A one-sided collective operation is one in which a single thread executes the collective operation and manipulates the shared memory partition of all the other participating threads.
The `upc_forall` has been tested only for the default block size. Future work can concentrate on testing `upc_forall` for different block sizes. The performance of `upc_forall` can also be tested with varying workloads of different sizes.

The collectives have been tested in the micro-benchmark suite with only one out of the nine possible combinations for the last argument. Some more combinations which might be useful in real time might be tested and a comparison between the different arguments could be made.

Testing the performance of UPC in machines like HPCx, which is an SMP cluster, should be interesting especially because of the architecture of the machine. The inter-node and intra-node performance can be measured and analysed. In the future as we are expecting to have multi-core machines, executing the benchmark in HPCx should help in understanding the behaviour of UPC in future multi-core architectures.

It would be useful to find out the overhead associated with the `upc_memput` and `upc_memget` functions as most of the collective operations involve copying of data internally for which one of these two functions are used.
Bibliography


