Computer Science Technical Report

High Performance Unified Parallel C (UPC) Collectives For Linux/Myrinet Platforms

Alok Mishra and Steven Seidel

Michigan Technological University
Computer Science Technical Report
CS-TR-04-05
August, 2004

MichiganTech

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Contents

1 Introduction **6**

1.1 Unified Parallel C (UPC) 6

1.2 The MuPC Runtime System 7

1.3 Myrinet and GM 8

1.4 Motivation 8

2 Message passing With GM **10**

2.1 GM ports and connections 10

2.2 GM communication protocol 10

2.3 DMA allocation schemes in GM 11

2.4 Sending messages in GM 12

2.5 Receiving messages in GM 13

3 Collective Communication Operations **15**

3.1 Need for Collective Communication Operations 15

3.2 Push, Pull and Naive Implementations 16

3.3 Collective Communication Operations in UPC 19

3.3.1 upc_all_broadcast 19

3.3.2 upc_all_scatter 19

3.3.3 upc_all_gather 20

3.3.4 upc_all_gather_all 20

3.3.5 upc_all_exchange 21

3.3.6 upc_all_permute 22

4 Past Work **23**

4.1 Reference Implementation 23

4.2 Setting up the communication library 23
4.3 Broadcast algorithms 24
4.4 Motivation for creating the synthetic benchmark 24
4.5 Parameters used in the testbed 25
4.5.1 Hardware Platform 25
4.5.2 Runtime System 26
4.5.3 Algorithms 26
4.5.4 Collective Operation 27
4.5.5 Number of threads involved 27
4.5.6 Message Length 28
4.5.7 Cache Length in the MuPC runtime system 28
4.5.8 Skew among threads 28
4.5.9 Synchronization within the collective 28

5 Current Work **30**
5.1 The GMTU collective communication library 30
5.2 DMA Memory Allocation Schemes 30
5.2.1 Memory registration 30
5.2.2 Dynamic Memory Allocation 31
5.2.3 Static Memory Allocation 32
5.2.4 Summary 33
5.3 Outline of collective communication functions in GMTU 33
5.3.1 GMTU `upc_all_broadcast` collective function 33
5.3.2 GMTU `upc_all_scatter` collective function 34
5.3.3 GMTU `upc_all_gather` collective function 36
5.3.4 GMTU `upc_all_gather_all` collective function 38
5.3.5 GMTU `upc_all_exchange` collective function 39

5.3.6 GMTU upc_all_permute collective function . . . 42
5.4 GM based synchronization . . . 43
5.4.1 Synchronization in GMTU for upc_all_broadcast . . . 44
5.4.2 Synchronization in GMTU for upc_all_scatter . . . 44
5.4.3 Synchronization in GMTU for upc_all_gather . . . 44
5.4.4 Synchronization in GMTU for upc_all_gather_all . . . 45
5.4.5 Synchronization in GMTU for upc_all_exchange . . . 45
5.4.6 Synchronization in GMTU for upc_all_permute . . . 45
5.5 The testbed and testscripts . . . 46

6 Packaging the library for initial release 47

6.0.1 Environment variables and flags for the GMTU library . . . 48

7 Results 51

7.1 Implementation of the testbed and testing details . . . 51
7.2 Testmatrix and testcases . . . 53
7.3 Naive push and pull based comparisons . . . 55
7.3.1 Comparison of GMTU algorithms . . . 57
7.4 Comparison of collective communication libraries . . . 63
7.4.1 upc_all_broadcast . . . 64
7.4.2 upc_all_scatter . . . 67
7.4.3 upc_all_gather . . . 70
7.4.4 upc_all_gather_all . . . 73
7.4.5 upc_all_exchange . . . 76
7.4.6 upc_all_permute . . . 79
7.5 Effect of Computational Time on Collective Performance . . . 82
7.5.1 Short computation . . . 82

7.5.2 Unequal computation 85

8 Conclusion **88**

Abstract

Unified Parallel C (UPC) is a partitioned shared memory parallel programming language that is being developed by a consortium of academia, industry and government. UPC is an extension of ANSI C. In this project we implemented a high performance UPC collective communications library of functions to perform data relocalization in UPC programs for Linux/Myrinet clusters. Myrinet is a low latency, high bandwidth local area network. The library was written using Myrinet's low level communication layer called GM.

We implemented the broadcast, scatter, gather, gather all, exchange and permute collective functions as defined in the UPC collectives specification document. The performance of these functions was compared to a UPC-level reference implementation of the collectives library. We also designed and implemented a micro-benchmarking application to measure and compare the performance of collective functions. The collective functions implemented in GM usually ran two to three times faster than the reference implementation over a wide range of message lengths and numbers of threads.

1 Introduction

1.1 Unified Parallel C (UPC)

Unified Parallel C (UPC) is a partitioned shared memory parallel programming language model that is being developed by a consortium of academia, industry and government. It is an extension of ANSI C, aimed for high performance computing on various platforms. UPC is similar to shared memory programming languages such as OpenMP [**?**] and High Performance FORTRAN (HPF) [**?**], with additional features such as barriers, locks, and collectives etc. In UPC,

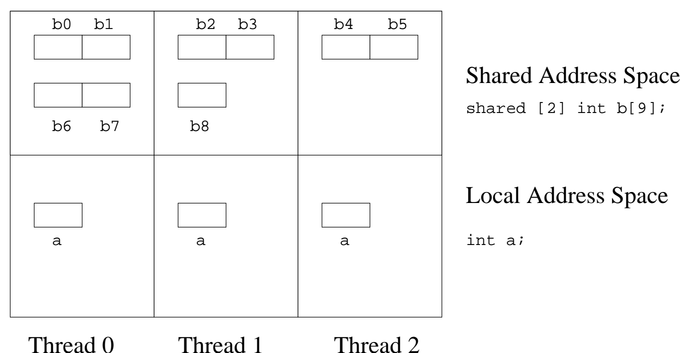


Figure 1. The partitioned shared memory model where threads have affinity to regions have the shared address space.

each participating thread is an independent process and has a shared address space along with a local address space. The concept of 'affinity' (Figure 1) emerges in the shared region of this programming model, where the 'ith block of the shared address region' is said to have 'affinity' to the 'ith thread'. The partitioned shared memory model allows for threads to access any portion of the shared address space, besides access to their own private address space. However, accesses to regions which do not have affinity to the particular thread maybe costlier than accesses to regions which have affinity to the thread. UPC allows the programmer to change the affinity of shared data to better exploit its locality through a set of collective communication operations. As mentioned earlier, besides traditional C features and constructs, UPC provides keywords and methods for parallel programming over the partitioned shared model. Example: shared keyword, synchronization mechanisms like Locks, Barriers and Memory consistency control (strict or relaxed). UPC is aimed to be a simple and easy to write programming language where remote reads and writes are done through simple expressions and assignments, respectively. This enables the user to concentrate more on the parallelization task, rather than worry about underlying architecture and communication operation as in message passing programming models where explicit reads/writes need to be posted for remote operations. The

local data are declared as per ANSI C semantics, while shared data are declared with the shared prefix (Figure 2) [?].

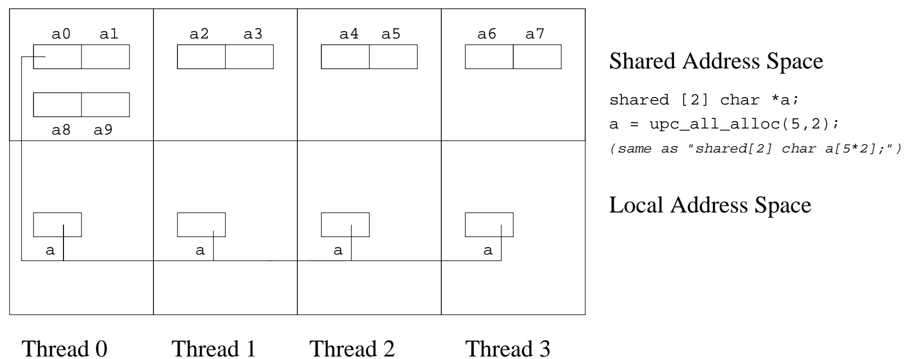


Figure 2. Creation and allocation of a shared array 'a' with block size `nbytes = 2`**, and size** `nblocks * nbytes = 10`**. Simply put, among 4 threads, the shared array of 10 elements is distributed in blocks of two elements per thread and wrapped around.**

1.2 The MuPC Runtime System

Originally UPC was developed for the Cray T3E. The popularity of cluster based, high performance computing required new compilers, runtime systems and libraries to be developed for extending the UPC language to this architectural model. The MuPC [?] runtime system, developed at Michigan Technological University (MTU), and the Global Address Space Network (GASNET) [?] runtime system for UPC, developed by University of California at Berkeley, are two such examples.

MuPC relies on MPI's communication library to implement read and writes to remote memory. Depending on the MPI implementation, we can choose the underlying interconnect. For example, the 16 node Linux cluster at MTU's Computational Science and Engineering Research Institute (CSERI) has both Ethernet and Myrinet interconnects. MPI programs can be run under either Lam-MPI or MPICH-GM to use either Ethernet or Myrinet interconnect, respectively.

A user can write supporting libraries for UPC just as in C, for mathematical calculations, string manipulation, etc. However, MPI library functions, such as MPI_Send(), MPI_Recv(), MPI_Broadcast(), cannot be used in UPC applications under the MuPC runtime system. This is because MuPC uses two pthreads, one for the user's UPC code and the other for communication, and due to the lack of thread safety in pthreads the user cannot use MPI calls in their UPC programs.

1.3 Myrinet and GM

The Myrinet [?] is a low-latency, high bandwidth local area network technology developed by Myricom. Myrinet interconnects with the PCI64B cards have a latency of 8.5us for short messages and upto 340 M Bytes/s two way data for large messages [?]. Compared to conventional networks such as Ethernet (100M Bytes/s), Myrinet provides features for high-performance computing using full-duplex links, flow control and cut-through routing. Myrinet also has certain functions that bypass the operating system to provide faster data transfer rates. It is a robust, highly scalable interconnect system and comes with a low-level communication layer from Myricom called GM.

The GM communication layer consists of a Myrinet-interface control program (MCP), the GM-API and features that allow communication over a Myrinet connected cluster. GM provides a notion of logical software ports distinct from Myrinet's hardware ports. The ports are used by process or client applications to communicate with the Myrinet interface directly. There are 8 such logical ports out of which ports 0, 1 and 3 are used internally by GM and ports 2, 4, 5, 6 and 7 are available to the user application. The maximum transmission unit (MTU), which defines the size of the largest packet that can be physically sent, is 4K Bytes for GM. During transmission all packets of size larger than the GM's MTU are broken down to around the 4K Byte size before being sent. The GM API provides various functions to send and receive messages, over Myrinet, using the communication layer. These functions can be used to develop middleware over GM, such as the MPICH-GM implementation of MPI.

Libraries developed over GM can use features such as OS-bypass to provide better performance. OS-bypass is a technique used by GM applications, which avoids calls to the operating-system for allocating and registering memory for sending/receiving by doing it once during initialization. After this the MCP and GM based applications communicate through the registered/allocated memory region for communication.

1.4 Motivation

High performance, cluster computing relies on high speed interconnects, such as Ethernet and Myrinet, that form their backbone. The Unified Parallel C (UPC) language, is a partitioned shared model based parallel programming language that uses runtime systems such as MuPC and GASNet on clusters. The MuPC runtime system, compiled under MPI libraries such as LAM-MPI and MPICH-GM, can use both the Ethernet and Myrinet interconnects, respectively.

Collective communication operations such as broadcast, scatter, gather, exchange, etc provide means of distributing data among processes in message passing based parallel applications. In MPI, for example, users can develop their own collective communication libraries using

MPI send and receive functions, but more efficient implementation of these collective operations are provided with the MPI library in the form of `MPI_Broadcast()`, MPI_Scatter(), MPI_AlltoAll(), etc [**?**].

As mentioned previously, in a partitioned shared memory model parallel programming language, such as UPC, accesses to shared data by processes without affinity to the shared data are translated to remote accesses. In UPC, users can perform data relocalization using shared assignments, however this process becomes highly inefficient as multiple assignment operations translate to as many remote accesses resulting in performance degradation (Figure 22). Therefore, collective communication operations are needed to perform data relocalization more efficiently and were initially implemented for UPC in the reference implementation [**?**] using the `upc_memcpy()` function. Although, this straight-forward implementation of the collective communication operations provided performance improvement over the naive version, a high performance implementation of collective communication operations was still desired.

Therefore, our goal was to develop a UPC collective communication library over Myrinet to provide a high performance implementation compatible with the UPC runtime systems, such as MuPC and GM based GASNet. We developed this library as middleware over GM using C and UPC commands. We also developed a synthetic benchmarking application, a collectives *testbed*, in UPC for emprical analysis of collective communication libraries on different runtime systems.

In the following sections we introduce the GM message passing model [**?**], followed by a brief overview of the collective communication functions in UPC. We then discuss our implementation of the testbed and the collective communication library. This library consists of GM implementations of broadcast, scatter, gather, gather all, exchange and permute. Performance results are then given for the various test conditions.

2 Message passing With GM

2.1 GM ports and connections

GM is a low-level message passing system that uses the Myrinet interconnect to send/receive messages over a connected network. At the application level, the communication model in GM is *connectionless* (Figure 3)[**?**]. Meaning, unlike other systems that use protocols to set up a communication link prior to message passing, such as the *request-reply protocol*, no handshaking is required by GM applications. A user-process can simply send a message to any node on the network, assuming a physical connection between the two nodes exists. The Myrinet switch maintains a routing table, with paths connecting all the active nodes in the network. As a result, communication operations between GM applications do not require route discovery protocols.

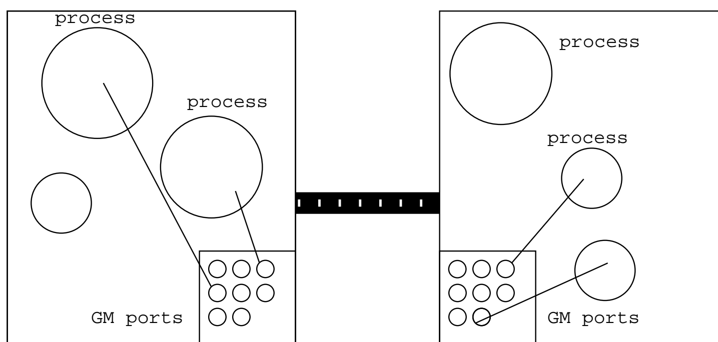


Figure 3. Ports and connections in GM. The messaging is connectionless in GM, meaning no route discovery needs to be done by a sending host - it simply says 'send this message to this node id through my this port' and the message is sent.[?]

2.2 GM communication protocol

GM applications use a message passing communication model similar to MPI. This model requires that every send operation on a source node have a matching receive operation at the destination node. The send and receive calls must also match in buffer size and message priority types at both ends. This serves as a tag to help to distinguish incoming messages faster. For example, while sending `nbytes` of data the sender passes `gm_min_size_for_length(nbytes)` as the size parameter and `nbytes` as the message length. The receiver does a `gm_provide_receive_buffer()` with the size parameter `gm_min_size_for_length(nbytes)` to receive the message.

It is also possible to do *remote puts* using remote DMA (RDMA) writes and *remote gets* using remote DMA (RDMA) reads in GM. While RDMA gets are available on GM-2 only, only RDMA puts are implemented over GM-1. Our work was done using GM-1 and therefore we were limited to using RDMA puts only. A remote put is essentially a one-sided send; the receiver does not need to post an explicit receive (or provide a receive buffer). Without remote reads (gets), GM clients can only implement *Push-based* data transfers in which the source sends the data to the destination(s). However, when RDMA reads are available it will be possible to implement *Pull based* functions, where receiving nodes simply read from a remote source location without requiring the source to send them a message.

All send and receive operations in GM are conducted using tokens to regulate and track messages. During initialization a GM client has a fixed number of send and receive tokens (Figure 4). A call to a send function releases a send token, which is returned when the send completes. Similarly, a call to `gm_provide_receive_buffer` will release a receive token and once a matching receive (based on size and priority) has been received.

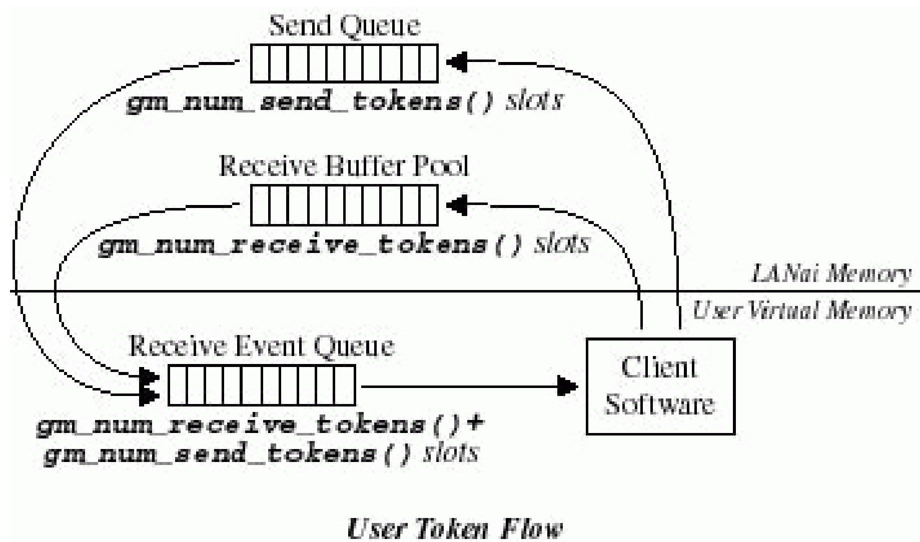


Figure 4. Tokens are used to keep track of sent/received messages by GM hosts. [*Source* : *Myricom* [?]]

2.3 DMA allocation schemes in GM

All messages sent and received by GM must reside in DMAable memory. GM allows clients to allocate new DMAable memory using calls to `gm_dma_malloc()` and `gm_dma_calloc()`.

There is also the facility to *register* existing memory regions in the user space through calls to gm_register_memory(). The process of registering memory makes the region non-pageable, this adds a page table entry to a DMAable page table that LANai accesses enabling GM to read/write onto that region. The process of deregistering memory, using gm_deregister_memory(), makes the region pageable again, and involves removing the memory address hash and pausing the firmware. This makes it more expensive to deregister memory than to register; our collective library uses this fact to provide further optimizations.

The process of dynamically allocating memory takes less time than registering memory, however it uses more memory as GM starts running a memory manager on top of the allocated regions. Therefore based upon memory or performance requirements, the user may choose one of the above memory allocation schemes in our collectives library.

2.4 Sending messages in GM

To send and receive messages the client application must keep a track of the number of send and receive tokens it has. Before making any calls to GM functions that require tokens it must make sure it possesses enough tokens for that operation. The GM send and receive protocol also uses *size* and *priority* tags. GM clients sending and receiving a message must both agree on the size and priority of the message. GM message priorities can be GM_HIGH_PRIORITY or GM_LOW_PRIORITY (Figure 5). The size of a message in GM is given by the function gm_min_size_for_length(nbytes), where nbytes is the length of the message in bytes [**?**].

A client application calls the GM send function by passing a *callback* function pointer and a *context* structure pointer to that function. The callback function is used to check the status of the send once the send completes using the parameters in context structure. A GM client sending a message gives up one send token which is returned when the send function completes and GM calls the callback function. The callback function is called with a pointer to the GM port of the sender, the context pointer of the send function and the status of send operation. The client application can use the callback function to verify that the send has completed successfully or not. The gm_send_with_callback() is used when the receiving client's GM port id is different from the sending client's GM port-id; otherwise gm_send_to_peer_with_callback is used since it is slightly faster.

It also is important to note that the sender does not to deallocate (or deregister) or modify the send buffer until the callback function returns successfully, until gm_wait_for_callback() completes, since the data then might become inconsistent.

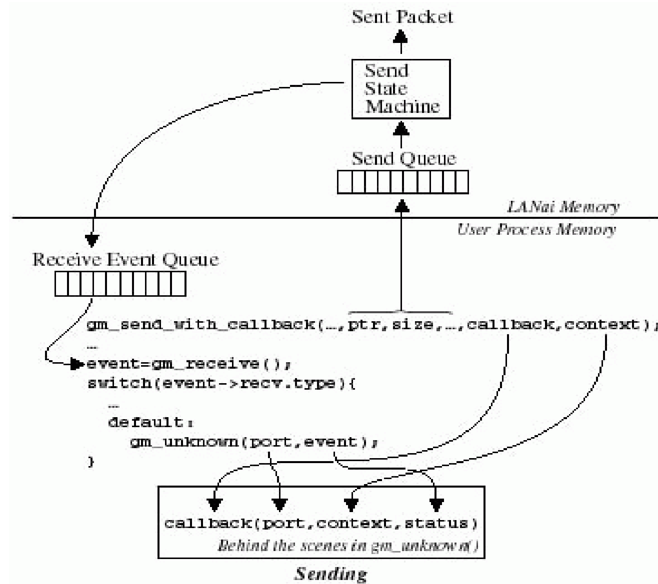


Figure 5. When sending a message in GM the message priority and size must be the same as that expected by the receiver.[*source* : *Myricom* [?]]

2.5 Receiving messages in GM

Receive operations in GM are also regulated by the notion of implicit tokens. Before a receiving client can receive a message intended for it, it must provide a receive buffer of the expected size and priority. A call to `gm_provide_receive_buffer()` is used and in the process the receiving client gives up a receive token. This token is returned after a receive event has been handled properly (Figure 6). Upon providing a receive buffer the receiving client checks for a `gm_receive_event()` using the `gm_receive_event_t` structure's `recv` type field. The receiver waits until a message of matching size and priority is received. When a message is received there are numerous types of receive event types that may be generated depending on the size, priority and receive port of the receive event type. The client application handles all the expected receive event types and the rest are simply passed to the `gm_unknown()` function. The `gm_unknown()` function is a special function that helps the client application resolve unknown messages types and allows the GM library to handle errors.

Depending on the gm_receive_event structure's receive type, the incoming message can be copied from the `recv` field's message parameter or the receive buffer provided by the user (`*_FAST_PEER_*` or `*_PEER_*`, respectively). We found that a message smaller than 128 bytes generated a `*_FAST_PEER_*` receive event and a larger sized message generated a `*_PEER_*`

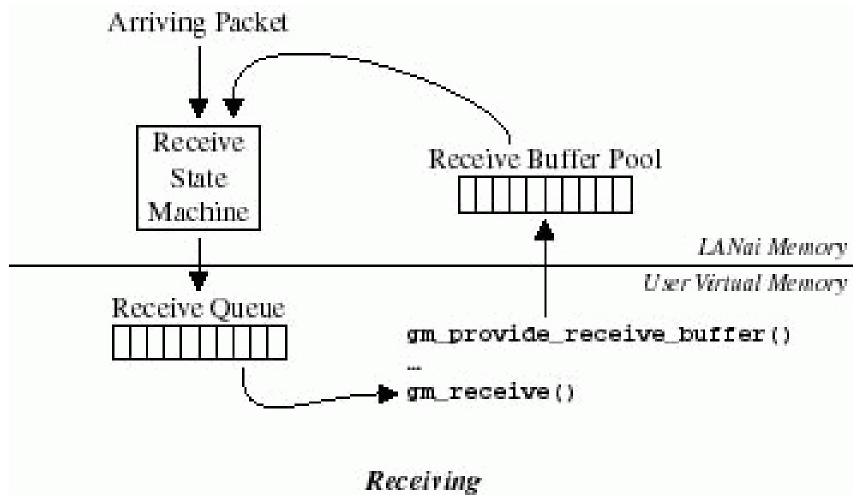


Figure 6. When receiving messages in GM, we must first provide a receive buffer of matching size and priority as the sender did while sending this message. Then we must poll for the total number of expected. [*Source* : *Myricom* [?]]

receive event. Similarly, a message with a high priority tag will generate a `*_HIGH_PEER_*` event and a low priority message tag will generate a `*_LOW_PEER_*` event.

A message is from a peer when the sender's GM port id and the receiver's GM port id are the same, otherwise it is simply a `*_FAST_*`, `*_ HIGH_*` or `*_LOW_*` instead of `*_FAST_PEER_*`, `*_ HIGH_PEER_*` or `*_LOW_PEER_*`. It is up to the application developer, to use the messages types appropriately in the receive functions.

3 Collective Communication Operations

3.1 Need for Collective Communication Operations

As mentioned earlier, the concept of affinity in UPC relates to partitions of shared address space being closer to a certain process (or thread) [?]. The effect of this is important in platforms such as Beowulf clusters, where the read/writes to areas of shared memory that processes do not have affinity to are costlier because they access remote memory (Figure 7). The cost increases if these accesses are frequently made by applications, and hence the need to re-localize arises. Collective operations provide a way to achieve this relocalization of data, resulting in a change in the affinity of the data item that leads to less costly accesses. An example of how the broadcast function performs relocalization is illustrated below:

- Pre-Broadcast: `b` has affinity to thread 0 only, any computation involving other threads would be costlier as they would be translated to remote read/writes

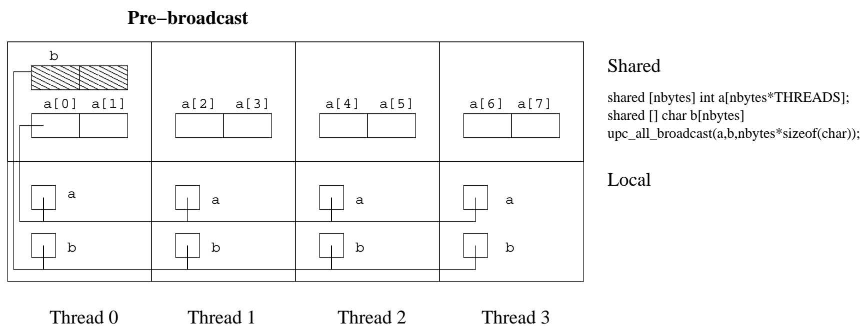


Figure 7. Referencing shared address space that a thread does not have affinity to, can be costly. This picture shows the `nbytes`-sized block of a char array `b` (shaded) that thread 0 has affinity to.

- Post-Broadcast: Each thread has a copy of ‘b’ in the shared area, ‘a’, with affinity to that thread after re-localization. As a result they can use ‘a’ instead of ‘b’ to reduce the cost of memory accesses to data contained in `b`

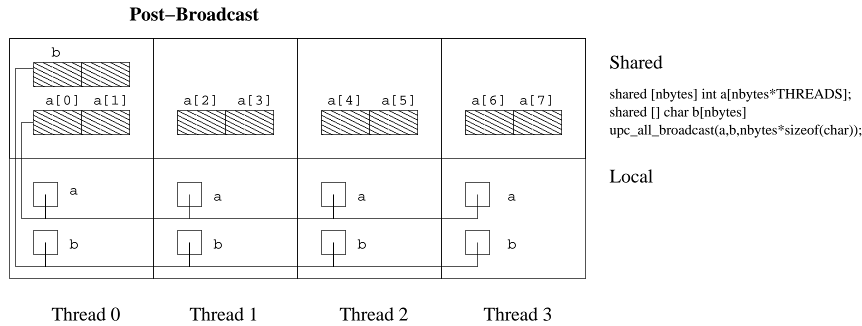


Figure 8. Post-broadcast we see that a copy of `nbytes` bytes of char array `b` have been sent to each thread and are stored in array `a`.

3.2 Push, Pull and Naive Implementations

There are two basic ways in which relocalization operations can be carried out, depending on which threads do the work. Consider a producer-consumer arrangement, where one producer produces goods that are consumed by many consumers. We can look at various aspects of this arrangement, but for our example let us consider how data is distributed. The questions we must ask are, *Does the producer bring the data to each consumer?* or, *Do consumers go up to the producer and demand their data?* In terms of hosts, nodes, blocks of shared memory and collectives, we can restate the above questions as: during a broadcast, *Does the source write the data to the others?* or, *Do the destinations read the data from the source?*

Therefore, collective communication operations where source thread(s) send the data to the destination thread(s) are regarded as *push* implementations, and operations where the destination thread(s) read the data from the source thread(s) are termed the *pull* implementations. Figure 9 shows how push and pull broadcast operations differ. In Figure 9(a) thread 0 is pushing the data to the destination threads by copying it into their memory locations. In Figure 9(b) each destination thread is responsible for pulling the data from thread 0's memory location.

The reference implementation [**?**] provides a straight forward way of performing both push and pull relocalization operations for all the collectives mentioned in the collective specification document [**?**]. The collectives in the reference implementation are implemented using the `upc_memcpy()` function which, in the partitioned shared memory model, is translated into remote gets or puts depending on whether it is the source or the desitnation location that is remote to the calling thread. An example of the reference implementation `upc_all_broadcast` collective function, in push and pull versions is given below.

- Psuedocode for push reference implementation of `upc_all_broadcast`

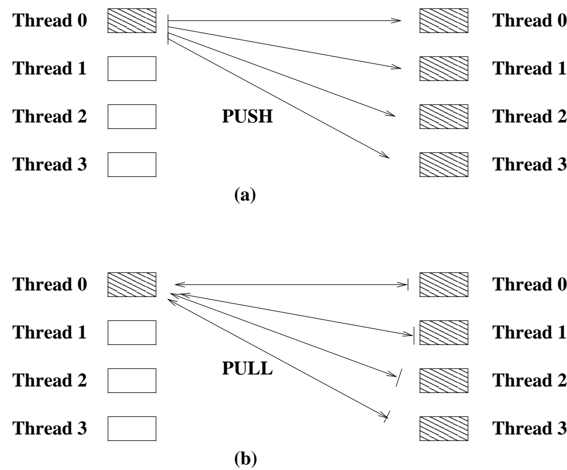


Figure 9. `upc_all_broadcast` **using 'push' and 'pull' based protocols. (a) Push based collectives rely on one thread copying the data to the rest, while (b) pull based collectives are more efficient and parallelized as destination threads are responsible to copying their own data from the source.**

```
begin upc_all_broadcast(shared void *dst,
                        shared const void *src,
                        size_t nbytes,
                        upc_flag_t sync_mode)
source:= upc_threadof 'src' array
if(MYTHREAD = source) then
 for i:=0 to THREADS
   upc_memcpy 'nbytes' from 'src' into 'dst+i'
 end for
end if
end upc_all_broadcast
```

- Psuedocode for pull reference implementation of `upc_all_broadcast`

```
begin upc_all_broadcast(shared void *dst,
                        shared const void *src,
                        size_t nbytes,
                        upc_flag_t sync_mode)
  upc_memcpy 'nbytes' from 'src' into 'dst+MYTHREAD'
end upc_all_broadcast
```

A UPC programmer can also implement the collectives in a *naive* manner using array assignments to perform data movement. In the partitioned shared memory model each of the assignment statements would translate to a remote get or remote put depending on the nature of the implementation.The overhead for such an implementation is therefore quite high since moving `n` elements would translate to as many remote operations. The MuPC runtime system with a software runtime cache based, allevates some of this cost; however this is highly depended on the cache size. For increasing number of elements the performance of the naive algorithm would degrade progressively. In our results we compare a naive implementation of the `upc_all_broadcast` collective against the reference implementation and against our library to confirm these observations.

- Psuedocode for the naive push reference implementation of `upc_all_broadcast`

```
begin upc_all_broadcast(shared void *dst,
                        shared const void *src,
                        size_t nbytes,
                        upc_flag_t sync_mode)
source := upc_threadof 'src' array
if(MYTHREAD = source) then
 for i:=0 to THREADS
  for j:=0 to nbytes
   n := i*nbytes+j
   set the n'th element of 'dst'
    equal to the j'th element of 'src'
  end for
 end for
end if
end upc_all_broadcast
```

- Psuedocode for the naive pull reference implementation of `upc_all_broadcast`

```
begin upc_all_broadcast(shared void *dst,
                        shared const void *src,
                        size_t nbytes,
                        upc_flag_t sync_mode)
for j:=0 to nbytes
  n := MYTHREAD*nbytes+j
  set the n'th element of 'dst'
    equal to the j'th element of 'src'
end for
end upc_all_broadcast
```

3.3 Collective Communication Operations in UPC

In this project we implemented six of the collective communication operations as specified in the *UPC Collective Operations Specification V1.0* [**?**]. The details of these collective functions, their parameters, and their operations on shared arrays are described below.

3.3.1 upc_all_broadcast

```
void upc_all_broadcast(shared void *dst, shared const void *src, size_t nbytes,
upc_flag_t sync_mode)
```

Description:
The src pointer is interpreted as:
`shared [] char[nbytes]`
The dst pointer is interpreted as:
`shared [nbytes] char[nbytes*THREADS]`
The function copies nbytes of the src array into each nbyte-block of the dst array.

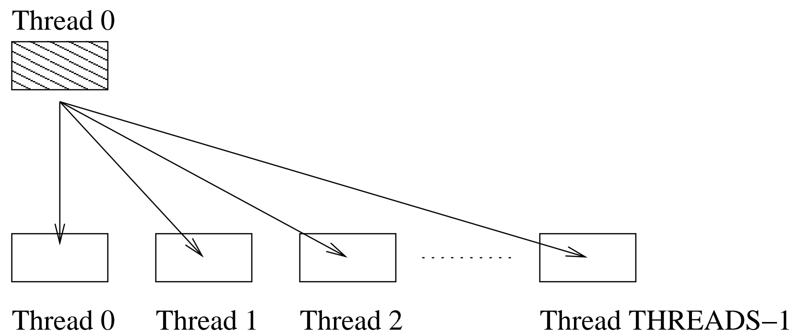


Figure 10. upc_all_broadcast push implementation

3.3.2 upc_all_scatter

```
void upc_all_scatter(shared void *dst,shared const void *src, size_t nbytes,
upc_flag_t sync_mode)
```

Description:
The src pointer is interpreted as:
`shared [] char[nbytes*THREADS]`
The dst array is interpreted as declaring:
`shared [nbytes] char[nbytes*THREADS]`

The ith thread copies the ith nbyte-block of the src array into the ith nybte-block of the dst array which has affinity to the ith thread.

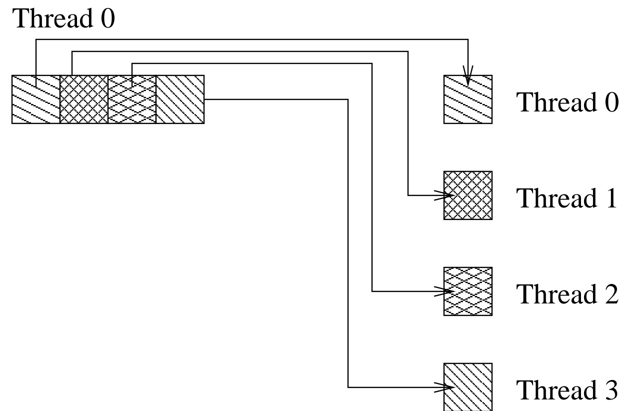


Figure 11. upc_all_scatter push implementation

3.3.3 upc_all_gather

```
void upc_all_gather (shared void *dst,shared const void *src, size_t nbytes,
upc_flag_t sync_mode)
```

Description:
The src pointer is assumed to be an array, declared as:
`shared [nbytes] char[nbytes * THREADS]`
The dst pointer is assumed to be declared as:
`shared [] char[nbytes* THREADS]`
The ith thread copies the ith nbyte-block of the src array, with affinity the ith thread, into the ith nybte-block of the dst array.

3.3.4 upc_all_gather_all

```
void upc_all_gather_all (shared void *dst,shared const void *src, size_t nbytes,
upc_flag_t sync_mode)
```

Description:
The src pointer is assumed to be an array, declared as:
`shared [nbytes] char[nbytes*THREADS]`
The dst pointer as:
`shared [ nbytes*THREADS] char[nbytes*THREADS *THREADS]`

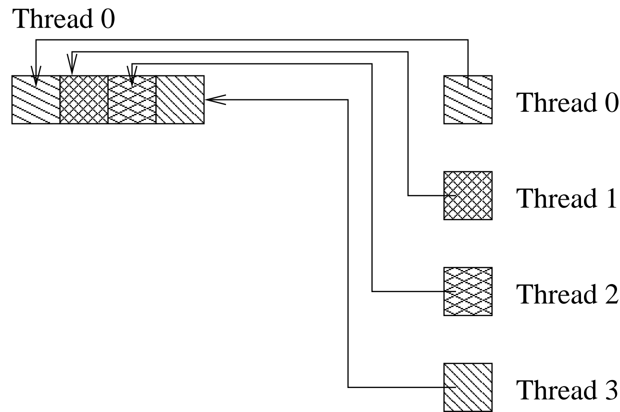


Figure 12. upc_all_gather push implementation

The ith thread copies the ith nbyte-block of the src array into the ith nybte-block of the dst array.

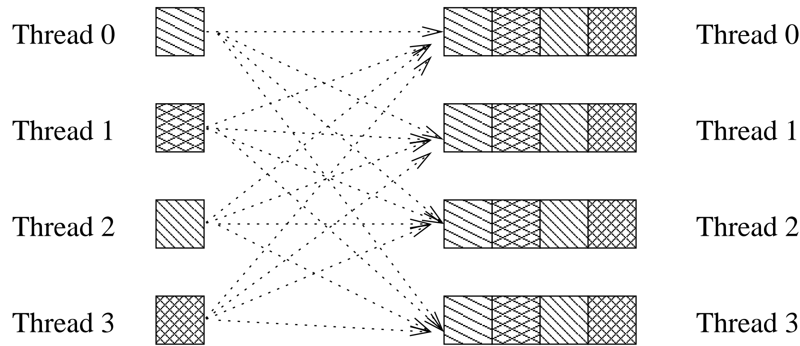


Figure 13. upc_all_gather_all push implementation

3.3.5 upc_all_exchange

```
void upc_all_exchange (shared void * dst, shared const void *src, size_t nbytes,
upc_flag_t sync_mode)
```

Description:
The src and dst pointers are assumed to be arrays, declared as:
`shared [nbytes*THREADS] char[nbytes*THREADS*THREADS]`
The ith thread, copies the jth nbyte-block of the src array into the ith nybte-block of the dst array which has affinity to the jth thread.

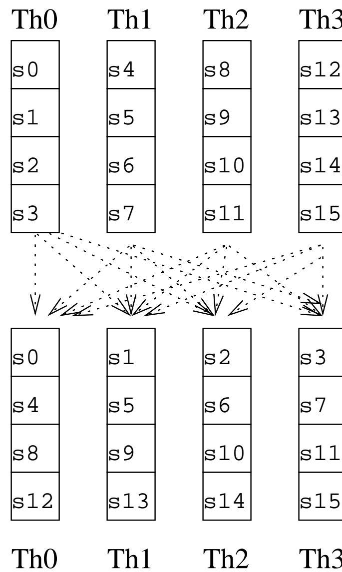


Figure 14. upc_all_exchange push implementation

3.3.6 upc_all_permute

```
void upc_all_permute (shared void *dst, shared const void *src, shared const
int *perm, size_t nbytes, upc_flag_t sync_mode)
```

Description:
The src and dst pointers are assumed to be char arrays that are declared as:
`shared [nbytes] char[nbytes*THREADS]`
The ith thread, copies the ith nbyte-block of the src array into the nybte-block of the dst array which has affinity to the thread corresponding to the ith element of the `perm` array.

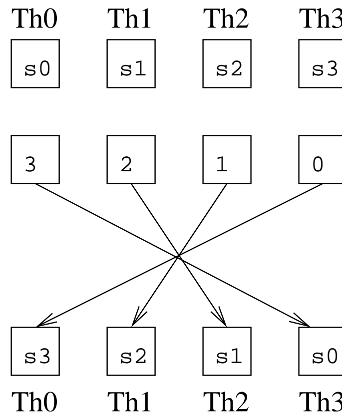


Figure 15. upc_all_permute push implementation

4 Past Work

4.1 Reference Implementation

The reference implementation of the collective library is a standard-conforming implementation but it does not necessarily perform efficiently. Our aim is to implement high performance versions of the collectives by understanding and using the GM message passing system.

4.2 Setting up the communication library

We started out by trying to understand programming in GM, through simple functions that implemented send/receive operations. In this first phase we used MPI to spawn processes. In these applications, message passing was not done using the usual send/receive functions provided by MPI, rather they used GM's send and receive functions.

Once we were able to establish a development base, we implemented similar message passing applications in the UPC environment to check for interoperability. At this point we also adopted the "`gmtu_`" nomenclature to identify and distinguish our library functions from those in GM.

We developed our UPC test applications on the MuPC runtime system.The MuPC runtime system was developed by PhD candidate Zhang Zhang, using Lam-MPI (over Ethernet) and MPICH-GM (over Myrinet). Depending on the version of the runtime system, we could choose the underlying interconnect for message passing in our UPC application and then compare its performance to those with our Myrinet-based library.

Our second development phase involved implementing the broadcast collective (`upc_all_broadcast`),

using different algorithms and send/receive protocols in GM.

4.3 Broadcast algorithms

We started developing the `GMTU` collectives library using GM for the `upc_all_broadcast` collective and implemented three different algorithms to perform this relocalization operation.

In the *straight push* broadcast the source thread sent the data to each thread one-by-one in `THREADS-1` steps. The *spanning tree* broadcast algorithm broadcast the data down a binary tree, rooted at the source, in `log(THREADS)` steps [**?**]. The RDMA put algorithm, called *Directed Receive Buffer (DRB) push*, used remote puts to push the data to the threads in `THREADS-1` steps.

The straight push and spanning tree push algorithms were initially implemented using a dynamic buffer and later using two static buffers (for sending and receiving). The DRB push algorithm relied on registering memory at the source and destination threads prior to data transfer (and synchronization), and after this the source push the data in `THREADS-1` steps to all the destination buffers.

We tested this initial library on the MuPC runtime system to check for interoperability of our UPC/C based GM library with the runtime system. This was done by developing a synthetic benchmark to simulate operating conditions of real parallel applications and testing our library (with broadcast) on it. We also compared the performance of our broadcast algorithms with the reference implementation and MPI broadcast algorithms, using the synthetic benchmark. We considered the reference implementation and MPI collectives as our baseline and used them for our comparsions.

4.4 Motivation for creating the synthetic benchmark

Standard benchmarks, like the Pallas MPI benchmarks (PMB), were not specifically constructed to measure and compare collective communication libraries like ours. The reason for this is that the computational time dominates the collective communication time by several magnitudes in these benchmarks (Figure 16). Our aim was therefore, to construct an application that will bring them to scale and our synthetic benchmark, or *testbed*, was developed with this goal.

The testbed interleaves computation and collective communication, to provide comparisons close to real world application. The time of computation is usually kept slightly higher than the collective communication time, to ensure all threads have completed one round of collective communication before entering the next round. The computation performed are all local,

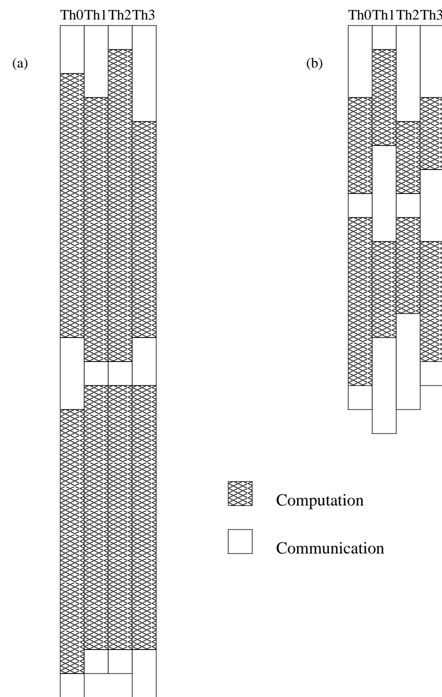


Figure 16. (a) Collectives as measured between computations in traditional benchmarks, computation and collective communication times are not proportional. (b) Collectives as measured by our benchmark, collective and computation times are kept proportional

there are no remote reads/writes during this period. During our final analysis, we vary the computation time and also take some measurements without any computation at all to study the effects of having computation.

4.5 Parameters used in the testbed

To begin the process of measurement we needed to identify the important parameters that would allow us to measure the performance of the library functions. Below is a list of parameters that we identified along with some discussion.

4.5.1 Hardware Platform

We tested our library on a 16 node Linux/Myrinet cluster. Each node has a dual-processor Intel Xenon chips with the i686 GNU Linux operating system and a 133 MHz PCI64B Myrinet/PCI

interface (LANai 9.0) running GM-1.6.5 on it. The cluster is also connected by a fast Ethernet connection.

4.5.2 Runtime System

The availability of different compilations of the MuPC runtime system (LAM MPI and MPICH-GM) provided us with different interconnect options. We could quantitatively compare our collectives against the reference implementation in both environments but we use the MPICH-GM distribution of the MuPC runtime system so as to make the comparison even in terms of the interconnect used.

Therefore, to highlight the differences between the reference implementation and our implementation - we compare against the reference implementation executed over the MPICH-GM based MuPC runtime system (MuPC-v1.1) [**?**]. Also, we compare the reference implementation compiled with Berkeley's UPC compiler and executed in the GASNet (Global Address Space Net) runtime system [**?**]. Gasnet also uses GM and so we should see some of the GM push collectives perform the same as the reference implementation.

4.5.3 Algorithms

The testbed can be compiled by linking different versions of our library to compare our algorithms against each other. We have memory registration, dynamic DMA allocation and static DMA allocation versions of each collective function. We also have the naive push and pull `upc_all_broadcast` collective algorithm, along with the reference implementation algorithms.

The titles of the algorithms, as used in our results are provided below.

- `MuPC-Naive-PULL` - Pull collectives from the naive implementation executed on the MuPC runtime system
- `MuPC-Naive-PUSH` - Pull collectives from the naive implementation executed on the MuPC runtime system
- `Berk-Naive-PULL` - Pull collectives from the naive implementation executed on the GASNet runtime system
- `Berk-Naive-PUSH` - Push collectives from the naive implementation executed on the GASNet runtime system

- `MuPC-mPULL` - Pull collectives from the reference implementation executed on the MuPC runtime system
- `MuPC-mPUSH` - Push collectives from the reference implementation executed on the MuPC runtime system
- `MuPC-GMTU_PUSH` - Push collectives using the GMTU memory registration implementation, executed on the MuPC runtime system
- `Berk-mPULL` - Pull collectives from the reference implementation executed on the GASNet runtime system
- `Berk-mPUSH` - Push collectives from the reference implementation executed on the GASNet runtime system
- `Berk-GMTU_PUSH` - Push collectives from our GMTU library executed on the GASNet runtime system

4.5.4 Collective Operation

The testbed is designed to measure the performance of each collective in various environments. The collectives we measured are:

- `upc_all_broadcast`
- `upc_all_scatter`
- `upc_all_gather`
- `upc_all_gather_all`
- `upc_all_exchange`
- `upc_all_permute`

4.5.5 Number of threads involved

The number of processes, or UPC threads, varies from 2 to 16. We measure odd numbered threads to check that spanning tree algorithms perform correctly. Comparing graphs of the same algorithms, for a collective function under different numbers of threads provided vital information about the scalability of our algorithms. It was also an important measure, during development, because at 2 threads all push collective operations are similar in nature, where 1 thread pushes `nbytes` of data to another thread. We used this observation to check for inconsistent performance and bugs.

4.5.6 Message Length

Varying the message length provided us with the most basic performance graph. We observe the performance of collectives, in our testbed, starting at a message length of 8 bytes to 64KB. We also measure times for some message lengths that are not powers of two.

4.5.7 Cache Length in the MuPC runtime system

The latest release of the MuPC runtime system uses a runtime software cache. The cache line length is by default 1K, and there are 256 lines in the cache. We always maintain the cache length at 1K in our tests.

4.5.8 Skew among threads

The behaviour of collectives in an asynchronous environment is an important concern. The testbed allows a user to introduce a start-up skew before the collective tests begin. The testbed also allows a user to perform measurements using skewed computations times, as compared to uniform computation times across threads during the collective tests. In the uniform computation test runs, all threads compute for a time slightly longer than the collective communication time while in the skewed computation, one of the threads computes for a longer time than the rest. The thread computing longer could be any thread except the source thread in the next round since delaying the source would delay all the others threads too.

4.5.9 Synchronization within the collective

The synchronization flags in UPC collective communication operations can be used by a programmer to control the type of synchronization within a collective function. The `IN_*SYNC` and `OUT_*SYNC` flags, where '*' can be either *ALL*, *MY* or *NONE*; are used in combination to specify the synchronization before and after data transfer within a collective.

For example, in UPC a programmer can use the different synchronization types to broadcast two disjoint blocks of data

Example:

```
upc_all_broadcast(dst1, src1,IN_ALLSYNC,OUT_NOSYNC);
upc_all_broadcast(dst2, src2,IN_NOSYNC,OUT_ALLSYNC);
```

Threads completing the first call, in this fashion, do not wait to synchronize with other threads and proceed immediately to make the second call. All threads then synchronize at the end

of the second broadcast only. This can help programmers use the collective functions more effectively. Most of our comparisons with the reference implementation are in the `IN_ALLSYNC` and `OUT_ALLSYNC` modes, as these are the most commonly used synchronization flags.

Note that the collective functions in the GMTU memory registration implementation, must explicitly know that the memory has been registered; also due to the RDMA put send used in these functions, they need to know that the message has been delivered. Therefore, we combine the `IN_ALLSYNC` and `OUT_ALLSYNC` operations with the data transfer in such a manner that the former tells all threads that all other threads have registered their memory locations and the latter tells all threads that the data have been sent. This implies that even in the `*_MYSYNC` and `*_NOSYNC` modes, the above mentioned barriers are needed in the GMTU memory registration scheme.

On the other hand the dynamic and static DMA allocation algorithms do not require explict synchronization in the `*_MYSYNC` and `*_NOSYNC` modes since they use GM's token based message passing. This is because in the token based message passing the source thread knows that the message has arrived at the destination from the status of the callback function and similarly the destination thread knows that the message is available as the `gm_receive()` function returns a receive event.

5 Current Work

The synthetic benchmarking testbed and the initial implementation of the broadcast function provided us with a development and testing base. We then implemented broadcast, scatter, gather, gather_all, exchange and permute collectives in UPC using experiences from our previous work. The collective functions were implemented using the *memory registration*, *dynamic DMA allocation* and *static DMA allocation* algorithms. The details of this work are provided below and are followed by the results from the testbed using the parameters discussed in the previous section.

5.1 The GMTU collective communication library

The high performance UPC collective communication library for Linux/Myrinet, was implemented using the three buffer allocation techniques. The library allows the user to choose between the memory allocation schemes.

In operating systems where *memory registration* is allowed (Eg. Linux [**?**]) the user can set the environment variable `GMTU_REGISTER_MEMORY` flag on, to select the memory registration implementation in the GMTU collective library . When no such flag is set, the length of the static buffer specified in the `GMTU_BUFFER_LENGTH` flag is used to choose between *static buffer* or *dynamic buffer* implementations in the GMTU collective library. For example, setting the `GMTU_BUFFER_LENGTH` to zero would switch the library onto the dynamic buffer allocation mode. The following subsections describe these DMA memory allocation schemes in detail, and these are followed by descriptions of the different algorithms for each of the six collective functions.

5.2 DMA Memory Allocation Schemes

5.2.1 Memory registration

GM provides the facility for applications to *register* memory, which makes the region registered *non-pageable*. This region can be then used to send from and receive into directly without allocating a separate DMAable buffer. The advantage of using this is that it provides a *zero-copy* performance, as we no longer have to copy the data from user space onto the allocated DMA space while sending and copy it back while receiving.

In our library we were able to register the *shared* arrays and send data from them, without copying it onto a DMA message buffer. These were achieved through calls to `gm_register_memory` and a matching call to `gm_deregister_memory`. Similar to MPI, we reduce the overhead of

registering and deregistering memory during a collective call by setting up a *registration table* to keep track of all registered memory regions. For example, during the first call to a collective function we register the array and store its address on a table. When subsequent calls to the collective function with pre-registered arrays are made we skip the cost of registering the array again. The memory locations registered in this manner, are all deregistered at the end of the application during the call to `gmtu_finalize()`, where used locations are made pageable again with a call to `gm_deregister memory`.

The total amount of memory that can be registered is specified by GM when it is installed. In GM-1.5 it is 7/8th of the physical memory, and if the client application tries to register beyond this limit an error is generated. In our library we trap this error and report it to the user, at which point they can choose to use this scheme with fewer amount of registered memory or can simply switch to the other algorithms (For example: select dynamic buffering scheme by unsetting the `GMTU_REGISTER_MEMORY` flag and setting the `GMTU_BUFFER_LENGTH` flag to zero).

We allow the user to specify the size of the registration table (default is 256 elements), to optimize searching the table during registration. The user can set the size of the `GMTU_REG_TABLE_MAX_ELEMENTS` flag, based upon the number of unique memory addresses that need to be registered.

Also, using the memory registration process allowed us to implement the remote put based send more effectively. Upon registering the source and destination buffers (in shared address space), within a collective, the source thread (after a synchronization primitive) could proceed to write directly into the destination's buffer. This is possible since the source thread can inquire the destination thread's remote address location through the `upc_addrfield()` operation, which across a partitioned-shared-address space like in UPC translates to a valid address [**?**].

For example, in `upc_all_broadcast()` the source can send the data to the i'th destination by calling `gm_directed_send_to_peer_with_callback()` with `upc_addrfield((shared char*)dst+i)` as the remote memory reference to write to.

5.2.2 Dynamic Memory Allocation

In systems where memory registration is not an option and the amount of physical memory is large, the user can choose the dynamic memory allocation implementation in the GMTU collectives library. In this scheme, during the call to a collective operation, the required amount of memory is allocated on the fly and released at the end of the collective. Although the process of allocating and deallocating memory takes less time than registering and deregistering memory within the collective, the dynamic allocation scheme requires copying data into (and out of) the DMA buffers for sending and receiving messages which add additional overhead on the collective operation.

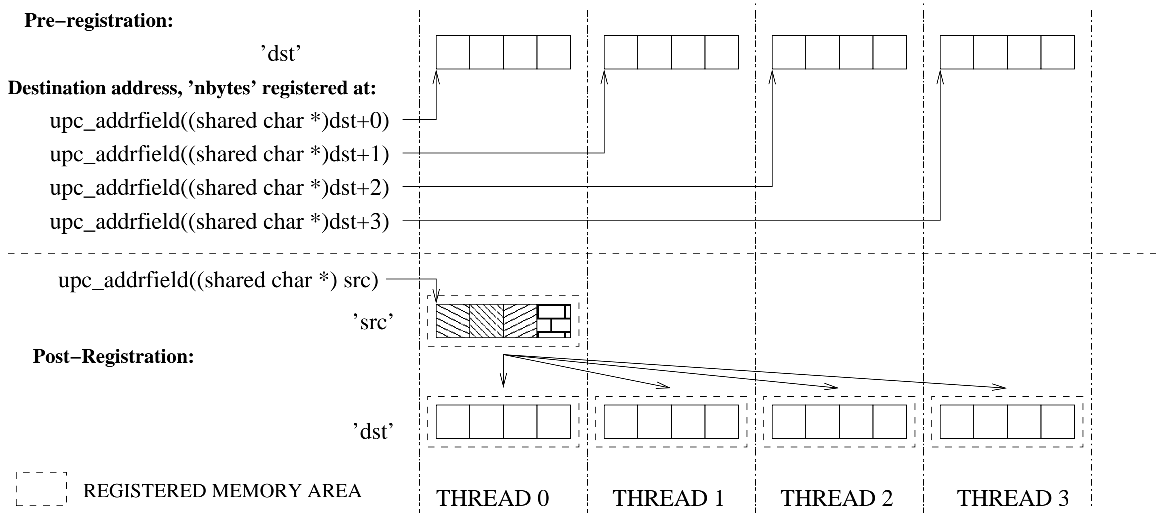


Figure 17. `upc_all_broadcast` **using memory registration, and rdma put for a push based implementation using GM**

5.2.3 Static Memory Allocation

The user also has the option of using the static buffer implementation in the GMTU collectives library. Here a send buffer and a receive buffer are allocated during initialization of the GMTU collectives. The length of the buffer is equal to the value of the `GMTU_BUFFER_LENGTH` variable as is specified by the user. The minimum value of the `GMTU_BUFFER_LENGTH` variable is 8KB and the maximum is 64KB.

The use of the static buffer imposes a restriction on the length of the data being sent and received. When the message size is larger than the buffer, our library packetizes the message into chunks big enough to fit into the buffer and the chunks are sent and received one-by-one. This constraint has an additional overhead imposed due to the packetization process. This is because while sending, the source needs to ensure that the previous data chunk has been received by the receiver(s) before it can overwrite the message buffer; and similarly, the receiver also needs to ensure that it has copied the data from its receive buffer before releasing a token for the next packet. The user can optimize this scheme by specifying a large enough static buffer size, for example: '`nbytes * THREADS`'[1] would be sufficient to avoid packetization.

[1] `nbytes` = largest expected message length

5.2.4 Summary

UPC collective communication operations implemented as a part of this project use three different memory schemes for send and receiving data in GM. The most efficient implementation is using memory registration in GM, as it reduces message copying overhead and also the need for allocating buffers for incoming and outgoing messages.

When the memory registration is not allowed, as in some operating systems, and the amount of DMAable memory is large; the dynamic memory allocation implementation can be used. Similarly the static memory allocated can be used when the amount of DMAable memory is limited and memory registration is not allowed. In the following sections we dicuss the algorithms used for each collective for all the three memory allocation implementations.

5.3 Outline of collective communication functions in GMTU

5.3.1 GMTU `upc_all_broadcast` collective function

- Broadcast using memory registration
  The source thread registers '`nbytes`' of the source array '`src`' and the destination threads register the region of the '`dst`' array they have affinity to `dst+MYTHREAD`. All threads then synchronize to tell the source that their buffers have been registered. This synchronization is present for all of the IN_*SYNC modes (ALL,MY and none). Once the destination threads have registered, the source begins sending the data.

  The algorithm is implemented as a spanning tree based broadcast, hence a destination thread waits until it receives the message from the source thread, and proceeds to retransmit the data to it's child nodes. This process continues until all threads have been served. There are `ceil[log(THREADS)]` steps in this algorithm.

- Broadcast using dynamic buffer
  Allocate all '`nbytes`' on source and destination nodes and send the data along the spanning tree as described above, using calls to `gm_send_to_peer _with_callback` and `gm_receive()`. In case of IN_ALLSYNC or IN_MYSYNC, threads allocate their DMAable memory first and then do the synchronization step.

- Broadcast using static buffer
  The static buffers for sending and receiving messages are `msg_buffer` and `recv_buffer` of length `GMTU_BUFFER_LENGTH`. The broadcast algorithm here too is the spanning tree based algorithm, as above, with the only difference being the limit on the length of message that can be sent.

If the buffer size is larger than the message length, the dynamic and static algorithms are similar; however, if the buffer is smaller then we packetize the message as described earlier and there are therefore `p * log(THREADS)` steps in all`p = ceil[nbytes * GMTU_MESSAGE_BUFFER_LENGTH]`.

- Discussion of algorithms used
  In the broadcast collective, there is a single source that sends the same `nbytes` of data from the '`src`' array to all the other threads. Using the memory registration scheme and the spanning tree algorithm, we can have the source perform the remote puts along the spanning tree, however we must ensure that the source threads for the subsequent rounds have the data before the can begin transmitting it. We avoid this overhead and use the regular send function where the receivers must post an explicit receive. (Figure 22).

  The dynamic and static buffer allocation schemes, use an indentical message passing method, similar to the memory registration scheme - the only difference being how the send/receive buffers are allocated. We were able to use the spanning tree in this collective, as compared to scatter or gather collectives, due to the nature and size of the data transmitted over each step of the spanning tree routine. The data items are always the same for all threads, and therefore only the required amount is transmitted - unlike the spanning tree based scatter, where in the `i`'th round, (`0< i < log(THREADS)+1`), '$\texttt{nbytes*THREADS/}2^i$' amount of data need to be sent (Figure 23) .

  Also special consideration has to been made for the static buffer allocation scheme, where the message size might be larger than the buffer size. In which case, sections of the source data, large enough to fit in the buffer, are sent in each round.

5.3.2 GMTU `upc_all_scatter` collective function

- Scatter using memory registration
  The source thread, in `upc_all_scatter` function registers '`nbytes * THREADS`' of the source array '`src`' and the destination threads register the region of the '`dst`' array they have affinity to `dst+MYTHREAD`. All threads then synchronize to tell the source that their buffers have been registered, and once the destination threads are ready the source thread calls the `gm_directed_ send_to_peer_with_callback()` with `upc_addrfield((shared char *) dst + i)` as the remote address of the destination buffer and sends `nbytes` starting from `(shared char *)src + nbytes * THREADS * i)` region of the shared address space, to the i'th thread.

  The algorithm is implemented as a straight push since a spanning tree would require allocating a larger receiver buffer for the nodes in the tree that serve as root nodes and are not the original source (see dynamic buffer based implementation for more details on this). There are therefore `THREADS-1` steps in this algorithm and during the send process

the destination nodes wait at the next synchronization step to know that the data has been sent (Example: OUT_ALLSYNC).

- Scatter using dynamic buffer
  The dynamic buffer based scatter is implemented using a spanning tree since all threads can allocate a 'nbytes*THREADS/2' sized buffer on for sending/receiving messages. This is the maximum amount of buffer space required because during the first step the source node (rank 0) sends the data to a thread that will be the source (in parallel) in the next step (Thread id = THREADS/2). The source node thus needs to send all of the data required by the destination node at that step, which is at max 'nbytes*THREADS/2'.

  Once all threads have allocated the buffer space, they follow the spanning tree based send/receive steps and the data is scattered in log(THREADS) steps.

- Scatter using static buffer
  In the static buffer scheme, depending on the available static buffer size (as specified by the user), our algorithm switches between the spanning tree based push and the straight push schemes. When '(nbytes * THREADS/2) > GMTU_BUFFER_LENGTH' we use the 'straight push' where p [2] packets are sent to each destination individually in 'THREADS-1' steps. There is an additional cost for waiting for callbacks for each destination per packet. Otherwise, when there is enough static buffer space available we send the data along the spanning tree in log(THREADS) steps as described in the 'dynamic buffer' based scatter above.

- Discussion of algorithms used
  The scatter collective, unlike broadcast, requires a large amount of buffer space for the spanning tree based collective. In the memory registration based scheme sending 'nbytes*THREADS/2^i' amount of data, for the i'th round (0< i < log(THREADS)+1), is no longer beneficial (like broadcast) - because we would be writing directly into the shared address space of the destination, which only posses nbytes of this space.

  We therefore avoid this situation by implementing a sequential push based scatter, for the memory registration based scheme, using the remote puts (Figure 24) . Also, the advantage of using the remote put with memory registration is that the sources does not have to wait for each destination thread to receive its data (unlike the static buffer scheme), it simply pushes the data out until it is done sending to all threads and then notifies them through a synchronization call (see GMTU based synchronization).

  The dynamic allocation scheme, on the other hand, can take advantage of the spanning tree based algorithm by having threads allocate buffers of size 'nbytes * THREADS/2', which is the maximum required buffer size for this scenario. However, the cost for

[2] p = ceil[nbytes/ (GMTU_MESSAGE_BUFFER_LENGTH)]

sending the nbytes of data for a large number of threads along the spanning tree, at max 'nbytes * THREADS/2', exceeds the cost of sending the data sequentially, always 'nbytes', in the long run (Figure 25). Therefore, the spanning tree based algorithm does not scale well with message length and thread size, and is not suited for an one-to-all-personalized communication scheme as scatter.

Considering the cost of performing the spanning tree based scatter 'p' [3] times, and using the above observation from the dynamic allocation scheme - we employ two different algorithms for the static buffer scheme. When the size of the static buffer is large enough to send 'nbytes * THREADS/2' bytes of data, we use the spanning tree algorithm. On the other hand, if the static buffer size is not large enough we use the straight (or sequential) push algorithm. The sequential algorithm suffers from the overhead of the source having to wait for each destination, before the next send operation to ensure that the buffer is clear - this is because the data items are distinct for each destination.

5.3.3 GMTU `upc_all_gather` collective function

- Gather using memory registration
  In memory registration based gather, the destination and source threads register 'dst' and 'src' regions. The source threads then wait for synchronization from the destination thread and once the destination thread is ready, the source threads call the `gm_directed_send_to_peer_with_callback()` with `upc_addrfield((shared char *)dst + nbytes * THREADS * MYTHREAD)` as the remote address of the destination buffer (corresponding to their thread id) and send nbytes starting from `(shared char *)src+ MYTHREAD)` region of the shared address space, to the i'th thread.

  The destination thread waits at the next synchronization point until all threads confirm that its data has been remotely put.

- Gather using dynamic buffer
  In the dynamic buffer allocation scheme for upc_all_gather, the destination thread allocates 'nbytes*THREADS' amount of DMAable memory, and frees 'THREADS-1' send tokens and polls for incoming messages from the other threads. The source threads simply allocate 'nbytes' of DMAable memory, copy their data and send it to the destination thread.

  Once the destination thread receives data from a source, it copies it from the DMA buffer onto the corresponding region of the 'dst' array in shared address space.

- Gather using static buffer
  In the static buffering scheme, the destination thread polls for 'THREADS-1 ' messages as

[3] `p = ceil[nbytes/ (GMTU_MESSAGE_BUFFER_LENGTH)]`

in the dynamic buffering scheme. In this case however, it does so 'p' [4] times to receive the entire message. Before receiving the data, the destination thread first sends a short message to the 'first' source thread (which is the next thread of the destination thread). This source thread sends its data, and once it is done sending the data to the destination thread it sends a short message to the next source thread.This indicates that the buffer at destination is available for another send. The subsequent source threads carry on this process of sending their data to destination, and then messaging the next source until all the threads are done sending to the destination thread.

- Discussion of algorithms used
  The gather collective, similar to the scatter collective, is unsuited for the spanning tree algorithm for large message lengths and thread sizes. However, the nature of this collective allows us to implement more efficient algorithms than in scatter. For example, a 'pull' based scatter is similar to a 'push' based gather for the work done by all threads. While scatter could benefit from remote get operations, available in GM-2, gather algorithms could use the remote puts available in the GM version we implemented the library in (GM-1.5).

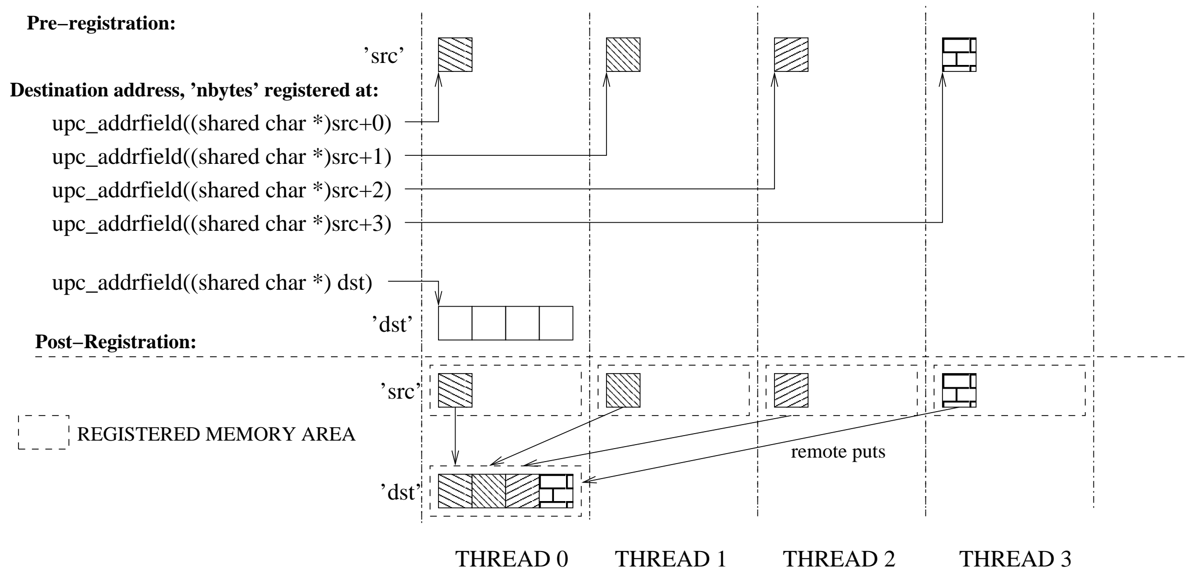


Figure 18. `upc_all_gather` **using memory registration and remote puts**

Therefore, all the source threads simply conduct a remote put into the region of the destination buffer, corresponding to their thread id, in the memory registration scheme (Figure 18). The remote puts are possible due to the shared memory model of UPC

[4] `p = ceil[nbytes/ (GMTU_MESSAGE_BUFFER_LENGTH)]`

which allows us to determine the address offset of the destination array on a remote thread(Figure 26)! In the absence of such an arrangement, GM applications would have to explicitly send the location of the remote destination array (via a scatter in this case, different offsets for different source threads).

The dynamic and static buffering schemes, therefore rely on the regular send/receive operations that GM provides to transfer data. In the dynamic scheme, the destination thread allocates a `nbytes * THREADS` sized receive buffer and simply waits for the source threads to fill it. When a receive event occurs the destination copies from the corresponding buffer into the matching source's region of the shared destination array. The other threads can write to their buffers in the same manner and there is no extra pre-caution required to avoid contention for the buffers(Figure 27). In the static buffer case however, we cannot do the same, and use a messaging scheme among the source threads to indicate that the buffer is available (as described in 'Gather using static buffer', above).

5.3.4 GMTU `upc_all_gather_all` collective function

- Gather all using memory registration
  In upc_all_gather_all, all threads broadcast '`nbytes`' of their data to all the other threads and also receive '`nbytes * THREADS-1`' data items from the others. Therefore, each thread registers `nbytes` of the the source array `src` and '`nbytes * THREADS`' of the destination array '`dst`' it has affinity to. All threads then wait for synchronization from Thread 0 and once they are aware that all the other threads are ready, all the threads call `gm_directed_send_to_peer_with_callback()` with `upc_addrfield( (shared char *)dst+i+(nbytes * THREADS * MYTHREAD))` as the remote address of the destination buffer and send `nbytes` starting from `(shared char *)src+ MYTHREAD)` region of the shared address space, to the i'th thread.

  To avoid congestion, during the '`THREADS-1`' send steps each thread calculates its partner in that step by XOR'ing their id with the step-number to obtain their partner's thread id. This way we avoid `THREADS-1` sends, say Thread 0, in the first step and then to Thread 2 in the next step etc.

- Gather all using dynamic buffer
  The dynamic buffer scheme for upc_all_gather_all uses a logical 'ring' to pass the data around. In this scheme, all threads calculate who their 'next' and 'previous' threads are and allocate '`nbytes`' of space for the send and receive buffers. Once all the threads are ready, each threads sends and receives for '`THREADS-1`' steps; where in the first step it sends the data it has affinity to - to its next thread and receives data from the previous thread. Once it receives the data and it's data has been received, each thread sends the

data it received on in the next round. This process sends 'nbytes * THREADS' data items around in a ring.

- Gather all using static buffer
  The static buffering scheme is very much like the logical ring based dynamic allocation scheme described above, with the difference being that the statically allocated send and receive buffers are used instead. When the message length is larger than the static buffer length, then there are p [5] packets sent in a total of 'p * THREADS-1' steps.

- Discussion of algorithms used
  The gather all collective is similar to all threads performing a broadcast of nbytes. This makes the collective better suited to the ring based algorithm, as described above. For the static and dynamic schemes, therefore, the data is transmitted along a logical ring for THREADS-1 steps (Figure 19(a)).

  The memory registration based gather all collective, on the other hand, is implemented to utilize the remote put function better(Figure 19(b)). Therefore, we have all threads simply put the data directly into the shared arrays of all the other threads (at an offset corresponding to their thread id), by using the remote addressing scheme described earlier(Figure 28).

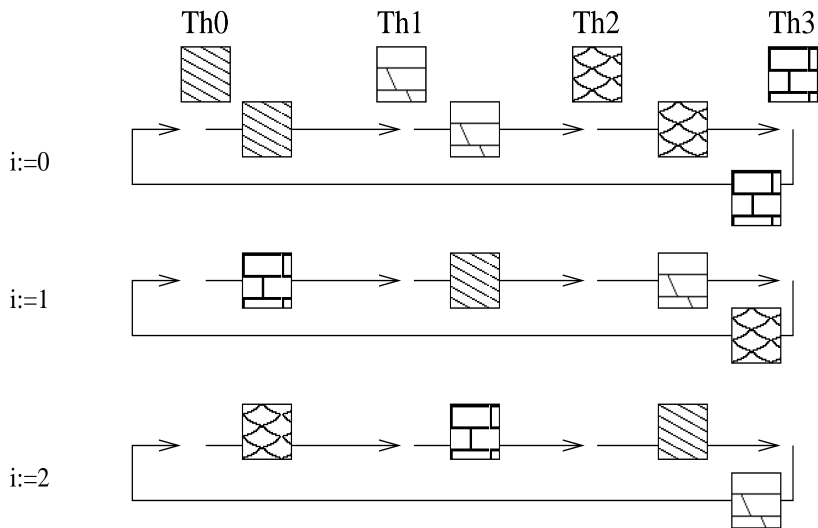
5.3.5 GMTU upc_all_exchange collective function

- Exchange using memory registration
  In the memory registration scheme, each thread registers 'nbytes * THREADS' of the source and destination arrays it has affinity to, that is (shared char *) src +MYTHREAD and (shared char *) dst +MYTHREAD respectively. Once all threads are aware that all the other threads are ready (post synchronization), they do THREADS-1 sends where similar to the upc_all_gather _all algorithm.

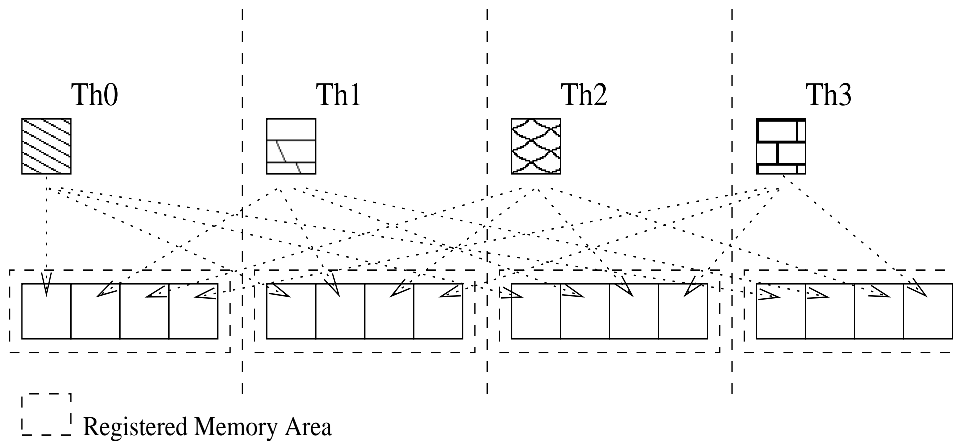
  Each thread sends to MYTHREAD XOR (i+1)'th partner (where 0 <= i <= THREADS-1), and thread 'i' sends to thread 'j' nbytes of data starting at (shared char *src)+i+ (nbytes * THREADS * j) by doing a remote put at the location upc_addrfield((shared char *dst)+j+ (nbytes * THREADS * i)).

- Exchange using dynamic buffer
  The process of exchanging data is similar to all threads performing a 'scatter', where each thread sends the i'th element of its source array to the i'th process. In the dynamic buffer based scheme therefore, unlike the memory registration scheme, all thread first allocate nbytes * THREADS of source and destination buffers - then copy data from the shared

[5] p = ceil[nbytes/GMTU_MESSAGE_BUFFER_LENGTH]



(a)



(b)

Figure 19. (a) using logical ring for dynamic and static DMA allocation based `upc_all_gather_all` **algorithms (b) Using remote puts, rdma write, for memory registration based** `upc_all_gather_all` **algorithm**

source array, they have affinity to, into their out buffer. All threads, release THREADS-1 receive tokens and then perform THREADS-1 sends, without waiting for a callback, since the buffer is distinct for each destination. Finally, they wait for THREADS-1 receives.

We decided against implementing the dynamic buffer based exchange using the XOR method (like in the memory registration based exchange), because we could avoid the cost of synchronization in the absence of one-sided-communication required for the send-receive based XOR algorithm. The freedom to allocate all the required memory at once, allowed us to use this alternative (unlike the static buffer based exchange - as below).

- Exchange using static buffer
  In the static buffer based exchange algorithm all threads, for 'i' steps (where `0 <= i <= THREADS-1`) copy their data onto the source buffer and send it to their partner (computed as `MYTHREAD XOR (i+1)`) in that step, and also receive the data into their receive buffer from their partner. When the static buffer size is smaller than the message length, there are p [6] such rounds of THREADS-1 exchanges.

  Unlike, the XOR algorithm in the memory registration based scheme, here each thread has to synchronize with its partner before sending its data by waiting for a short message from that partner. Threads provide a receive token in advance for the short synchronization message and the data message from their partner and send a short message to their partner. The send is a non-blocking send as the threads then poll for a data or a short message from their partners. If they receive a short message from their partner, they send the partner its data and wait for their data. If it is a data message on the other hand, then they copy it and wait for the short message from their partner.This extra bit of synchronization adds to the cost of this this algorithm.

- Discussion of algorithms used
  The exchange collective operation is similar to all threads performing a 'scatter'. An efficient implementation is the hypercube algorithm [**?**], where threads compute their partners and exchange their data in THREADS-1 steps. We use this where memory registration allows us to use the remote put operation (Figure 30). However, for the static and dynamic allocation schemes, where we are limited to using the regular send/receives in GM - the hypercube exchange algorithm proved difficult to implement.

  The token based send receive in GM required explicit synchronization before partners could exchange data, as described above. Due to this, in the dynamic (Figure 31) scheme we allocated `nbytes * THREADS` amount of receive buffer on all threads. The threads then release THREADS-1 receive tokens corresponding to a unique location in the buffer for each source, then send the data to all threads and wait for data to arrive from all

[6] `p = ceil[nbytes/GMTU_MESSAGE_BUFFER_LENGTH]`

threads. This is a an area of future investigation, where we would like to develop an optimal algorithm for the static and dynamic buffering scheme using GM's token based message passing, for the exchange algorithm.

In contrast we use the XOR based exchange for the static buffer scheme, due to the buffer limit, and observe that the performance of the dynamic buffer based algorithm is comparable to the static buffer algorithm.

5.3.6 GMTU `upc_all_permute` collective function

- Permute using memory registration
  The upc_all_permute collective uses a permute vector where `perm[i]` determines which thread receives data from the i'th thread. In the memory registration scheme, all threads first register `nbytes` of their source and destination arrays. After synchronization, they call `gm_directed_send_to_peer _with_callback()` with `upc_addrfield((shared char *)dst+ perm[MYTHREAD])` as the remote address of the destination buffer and send `nbytes` starting from `(shared char *)src+ MYTHREAD)` region of the shared address space, to the `perm[MYTHREAD]`'th thread.

  If the value of `perm[i]` happens to be `i` then the `i`'th thread does a local memcpy of the shared data using
  `memcpy( upc_addrfield ( (shared char *)dst + i), upc_addrfield( (shared char *)src + i), nbytes )`

- Permute using dynamic buffer
  In the dynamic buffer scheme, all threads allocate `nbytes` of send and receive buffers, release a receive token and copy their data on to the send buffer. They then send it out to `perm[MYTHREAD]`'th thread and wait to receive a data from their source. In case `perm[i]` happens to be `i` then the result is the same as described above in the memory registration scheme.

- Permute using static buffer
  The static buffer is similar to the dynamic buffer scheme except that the send and receive buffers are statically allocated and in case the message length is greater than the static buffer length, then there are 'p' [7] sends and receives per thread.

- Discussion of algorithms used
  The permute collective relies on straightforwad push for all three memory allocation schemes. In the dynamic (Figure 33) and static buffer schemes, the regular token based send/receive functions are used - where threads first release a receive token, then send

[7] `p = ceil[nbytes/GMTU_MESSAGE_BUFFER_LENGTH]`

their data to their destination and finally poll for an incoming message. In contrast, in the memory registration scheme, all threads simply put the data into their destination thread's shared array and wait at the next synchronization point. Once all threads cross this synchronization point, they can read the data in their shared destination array (Figure 32).

5.4 GM based synchronization

The collectives in the reference implementation rely on the `upc_barrier` for synchronization. A collective with IN_ALLSYNC and OUT_ALLSYNC based sync mode requiring two barriers could be further optimized by using GM based application level barrier.

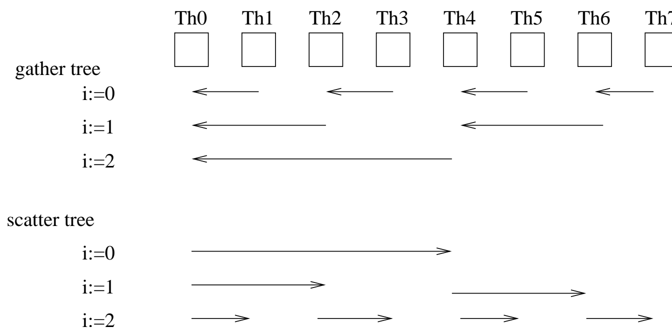


Figure 20. GM based barrier using gather and broadcast collectives for a short message length

We implemented a GM based barrier using a combination of spanning tree based gather followed by a spanning tree based collective (Figure 20), using a short message, for the MuPC runtime system. The idea was to combine the collective operation with the semantics of the `IN_ALLSYNC` and `OUT_ALLSYNC` synchronization modes. The synchronization scheme for each collective operation is outline below. The synchronization methods described, below, are used in all the three DMA allocation based algorithms described earlier, for each collective, and are only optimized for the 'memory registration' based algorithms. Our future work would include optimizing these for '`*_MYSYNC`' and '`*_NOSYNC`' modes in the dynamic and static buffer allocation algorithms.

Also we discovered that because the Berkeley GASNet runtime system also uses GM's messaging, we could not use our GM based synchronization for all situations. In cases where the `gm_min_size_for_length()` is the same for both the data and synchronization messages, we observed contention for tokens between GASNet runtime system's and our library's message.

Using the `upc_barrier()` based `IN_ALLSYNC` helped reduce the above problem, and our test

results for the GMTU library based collectives, compiled with Berkeley's UPC and executed over GASNet runtime system, use the native UPC based synchronization.

5.4.1 Synchronization in GMTU for `upc_all_broadcast`

`IN_ALLSYNC`, `IN_MYSYNC` and `IN_NOSYNC` is implemented as a spanning tree based gather, with the thread with affinity to 'src' (source thread) as the destination of the gather. The source thread therefore does not start sending the data until all destination threads are ready, while the destination threads message the source and poll on the `gm_receive()` function.

`OUT_ALLSYNC` is implemented as a spanning tree based gather, with the thread with affinity to 'src' (source thread) as the destination of the gather; followed by a spanning tree based broadcast by the source thread.

In `OUT_MYSYNC` and`OUT_NOSYNC` modes the source thread does a spanning tree based broadcast to indicate to the destination threads that the data is available.

5.4.2 Synchronization in GMTU for `upc_all_scatter`

`IN_ALLSYNC`, `IN_MYSYNC` and `IN_NOSYNC` is also implemented as a spanning tree based gather, with the thread with affinity to 'src' (source thread) as the destination of the gather. The source thread therefore does not start sending the data until all destination threads are ready, while the destination threads message the source and poll on the `gm_receive()` function.

When using the Berkeley runtime system, we used the native UPC barrier for all the three GMTU algorithms, to ensure that there is no contention for receive tokens due to the synchronization between our application and GASNet since both use GM.

`OUT_ALLSYNC` is implemented as a spanning tree based broadcast followed by a gather, with the thread with affinity to 'src' (source thread) as the source of the broadcast and the destination of the gather, respectively.

In `OUT_MYSYNC` and`OUT_NOSYNC` modes the source thread does a spanning tree based broadcast to indicate to the destination threads that the data is available.

5.4.3 Synchronization in GMTU for `upc_all_gather`

`IN_ALLSYNC` is implemented as a spanning tree based gather followed by a tree based broadcast, for the memory registration and dynamic allocation schemes for the MuPC runtime system based tests. In the case of the static buffer based gather, we use `upc_barrier()` for

IN_ALLSYNC, for the MuPC based test due to the nature of the algorithm which does not allow for a GM based synchronization.

In this collective too, when using the Berkeley runtime system, we used the native UPC barrier for all the three GMTU algorithms. It is interesting to note that only scatter and gather collectives are affected by the usage of a GM based synchronization for IN_ALLSYNC, over GASNet. Our future work involves resolving this issue.

IN_MYSYNC and IN_NOSYNC is implemented as a spanning tree based broadcast with the thread with affinity to 'dst' (destination thread) as the source of the broadcast. The source threads, therefore do not start sending the data until the destination thread is ready.

OUT_ALLSYNC, OUT_MYSYNC, OUT_NOSYNC is implemented as a spanning tree based broadcast followed by a gather, with the thread with affinity to 'dst' (destination thread) as the source of the broadcast and the destination of the gather, respectively.

5.4.4 Synchronization in GMTU for upc_all_gather_all

IN_ALLSYNC, IN_MYSYNC and IN_NOSYNC is implemented as a spanning tree based gather followed by a tree based broadcast with the thread with Thread 0 as the destination of the gather and the source of the broadcast. All the other threads, therefore do not start sending the data until they are sure all the other threads are ready.

OUT_ALLSYNC, OUT_MYSYNC, OUT_NOSYNC is implemented as a spanning tree based broadcast followed by a gather, with the thread with Thread 0 as the source of the broadcast and the destination of the gather, respectively.

5.4.5 Synchronization in GMTU for upc_all_exchange

IN_ALLSYNC, IN_MYSYNC and IN_NOSYNC is implemented as a spanning tree based gather followed by a tree based broadcast with the thread with Thread 0 as the destination of the gather and the source of the broadcast. All the other threads, therefore do not start sending the data until they are sure all the other threads are ready.

OUT_ALLSYNC, OUT_MYSYNC, OUT_NOSYNC is implemented as a spanning tree based broadcast followed by a gather, with the thread with Thread 0 as the source of the broadcast and the destination of the gather, respectively.

5.4.6 Synchronization in GMTU for upc_all_permute

IN_ALLSYNC, IN_MYSYNC and IN_NOSYNC is implemented as a spanning tree based gather

followed by a tree based broadcast with the thread with Thread 0 as the destination of the gather and the source of the broadcast. All the other threads, therefore do not start sending the data until they are sure all the other threads are ready.

`OUT_ALLSYNC, OUT_MYSYNC, OUT_NOSYNC` is implemented as a spanning tree based broadcast followed by a gather, with the thread with Thread 0 as the source of the broadcast and the destination of the gather, respectively.

5.5 The testbed and testscripts

Our next step was to develop the testbed and create test scripts to automate the testing and data collection process. The various parameters for the testbed made it possible to generate many different 'views' of the collective library being tested. Our goal was to identify all the possible combinations of parameters that would generate these views, and implement test scripts to separate test groups.

For example, the testscript allows a user to specify the collective function to be tested, along with the different collective libraries to be linked with the testbed, the various message lengths that each test for a collective is executed over and `THREAD` sizes for varying number of processes. Once this information is available the testscript generates an executable for each collective library specified, generates relevant testscripts and data files/directories. The testscript also creates gnuplot script files for each collective function, comparing the different implementations of a collective (implemented in the different libraries), for varying `THREAD` sizes.

The testbed, together with the associated testscript formed an integral part of our development process since we were able to rapidly visualize the implemented strategies and compare them with existing ones. Our next step was then to prepare the library and the benchmarking application for our intial release.

6 Packaging the library for initial release

Our collective communication library for UPC developed using GM is called the 'GMTU-UPC collective communication library'. In our initial release, the library shall comprise of the six collective communication operations implemented using the different DMA allocation schemes, along with the synthetic benchmark or testbed, the associated testscripts, and a Makefile to compile and link the libraries to the testbed and other UPC applications.

A 'README' file is also included describing the library usage, its environment flags, the testbed and testscripts. The initial release is organized in the following manner:

```
../GMTU_COLL/ver1.0/
->bin                  { The binaries from Makefile go here}
->etc                  { The 'machine file' should be here}
->include              { GMTU header file}
    gmtu.h

->lib
   upc_collective.o     { The reference implementation}
   upc_all_broadcast_gm.o
   upc_all_scatter_gm.o
   upc_all_gather_gm.o
   upc_all_gather_all_gm.o
   upc_all_exchange_gm.o
   upc_all_permute_gm.o

->test
 ->test_$date$
  ->upc_all_broadcast    { The test files are here for each algorithm }
  ->upc_all_scatter
  ->upc_all_gather
  ->upc_all_gather_all
  ->upc_all_exchange
  ->upc_all_permute
  testall.test

->data
 ->data_$date$
  ->upc_all_broadcast    { The data files are here for each algorithm }
  ->upc_all_scatter
```

```
  ->upc_all_gather
  ->upc_all_gather_all
  ->upc_all_exchange
  ->upc_all_permute

->testbed
  testbed.c              { The root testbed file }
  testbed_aux.c          { The aux testbed file with utilities}

->working
  ->gmtu_bin
     gmtu_callback_wait.c        { callback and wait functions}
     gmtu_local_send_receive.c   { local send/recv functions}
     gmtu_shared_send_receive.c  { shared send/recv functions}
     gmtu_sync.c                 { The gm-based sync functions}
     gmtu_util.c                 { GMTU utility functions}

   ->gmtu_collective
        upc_all_broadcast.c      { The implementations for all
        upc_all_scatter.c          collectives using Memory Reg,
        upc_all_gather.c           Dyn Alloc and Static Alloc
        upc_all_gather_all.c       algorithms}
        upc_all_exchange.c
        upc_all_permute.c

 README                          { The Readme file}
 Makefile                        { GMTU Makefile }
 generate_test_script.pl         { Test script generator}
```

6.0.1 Environment variables and flags for the GMTU library

The three different implementations of the collective operations in the GMTU library are chosen, as described in our 'Current Work' section, using environment variables. A list of these variables along with their description are provided below:

- GMTU_REGISTER_MEMORY
  The GMTU_REGISTER_MEMORY flag if defined allows the GMTU library to use the memory registration based algorithms. This flag must be set during initialization for the memory registration based implementation, as it allows the memory registration table to be set up.

When no such flag is set the GMTU_BUFFER_LENGTH variable is used to choose between the static and dynamic buffering modes.

- GMTU_BUFFER_LENGTH
  This variable specifies the size of the send and receive static buffers when its value is more than zero and the GMTU_REGISTER_MEMORY is turned off. When the value is more than zero, but less than the GMTU_MIN_BUFFER_LENGTH the size of the static buffer is set to the GMTU_MIN_BUFFER_LENGTH, which is fixed at 8K. The maximum size is specified by the GMTU_MAX_BUFFER_LENGTH which can be adjusted by the user.

  When the value of the buffer length is zero or less and the GMTU_REGISTER_MEMORY is turned off, the GMTU library uses the dynamic allocation scheme which allocates at most nbytes * THREADS amount of memory for any given collective. Therefore, setting the buffer length to zero implies that the user is certain that amount of DMA'able memory is atleast nbytes * THREADS, where nbytes is the size of the largest message used in the collective communication operation.

- GMTU_MAX_BUFFER_LENGTH
  This variable is used to check for the ceiling of the static buffer allocated, and by default is 64K. A user, wishing to allocate a static buffer higher than this default size should set this variable to the a new limit (higher or equal to the desired buffer size) first. This is useful in systems where the user is aware of the size of the DMA'able memory available.

- GMTU_REG_TABLE_MAX_ELEMENTS
  This variable is used to specifiy the size of the registration table in the memory registration mode of the GMTU collectives. The size by default is 256, and imples that as many new and distinct shared address locations can be registered. Registering a previously registered memory address, for a 'length' (in bytes) equal (or less) to the previously registered length results in a 'noop', however when the length is larger that the previously registered region length - the region is registered again. Re-registering in this fashion does not use up any extra location in the registration table, that is the same entry is registered but only the registration length changes.

  Therefore, the size of the table determines how many distinct regions in the user space need to be registered and made non-pageable. This size is not related to the 'length' or amount of memory registered and there is a possibility that the user may register more memory than is allowed, without using up all the table elements. In such a case, the library directs the user to choose between other memory allocation schemes (static or dynamic buffer).

- GMTU_DATA_PORT_NUM
  The GMTU_SYNC_PORT_NUM variable allows the user to choose which GM port should be

used for sending data messages. It is by default port 7, and when this port is not available on a certain thread, it will search for other available ports for messaging.

- `GMTU_SYNC_PORT_NUM`
  The `GMTU_SYNC_PORT_NUM` variable allows the user to choose which GM port should be used for sending synchronization messages. It is by default port 6, and when this port is not available on a certain thread, it will search for other available ports for messaging.

- `_USE_UPC_BARRIER_IN_` and `_USE_UPC_BARRIER_OUT_`
  The `_USE_UPC_BARRIER_IN_` and `_USE_UPC_BARRIER_OUT_` flags are used to choose the native `upc_barrier()`, instead of GMTU's message passing barrier, for synchronization before and after a collective call, respectively. The user may choose to use the native barrier in run time systems which use GM's messaging layer diretly, and where there is a possibility of conflict between the GMTU's synchronization messages and the runtime system's messages. For example, Berkeley's UPC also uses GM and we found that for certain collectives, the synchronization messages from our library conflicted with that of the runtime system, and using the native barrier for synchronization helped solve this problem.

7 Results

7.1 Implementation of the testbed and testing details

The collective communication library developed using GM, was compared against the reference implementation using the latest release of the MuPC runtime system, MuPC v1.1 compiled with MPICH-GM [?]and also using the Global Address Space Networking (GASNET) [?] runtime system for UPC. Each of the comparisions was conducted using the synthetic application benchmark, or testbed, developed in UPC. The various libraries compared were compiled for different `THREAD` sizes using MuPC's `mupcc` script and Berkeley's `upcc` script, to produce corresponding libraries.

These libraries were linked with the testbed to produce binaries, which were then used to test any of the collectives for the varying message lengths. For example, the binary `testbed_mPULL_2_TESTCASE_0` was used to test the *pull* based reference implementation (of all collectives) for 2 threads, for test case zero - which as per the test matrix (Figure 21) is the test for small computation time. A single test run, for a collective, a fixed message length and a fixed thread size, is executed 1000 times.

The testbed conducts the test runs, upon initialization, after calling a warm-up routine with a small number of calls to the collective being tested. This allows us to obtain a raw measure of the collective, with no computation, for determining the computation time in the actual test runs. During the test run, each call to the collective operation is followed by the computation which is conducted for the a pre-determined amount of time, dependent on the collective time measured during warm-up and a constant. The amount of computation can also be 'skewed' to make one thread compute longer, which is approximately 10 times the normal compute time. All of the computation is local, and there is no remote operation during the compute phase. This is done to *quiet* the network before measuring the communication time again. The format of the testbed is therefore:

```
procedure upc_mtu_testbed()

/* warm up */
for i:=0 to (Max warm up runs)
 start time
 call collective
 stop time
 warmup time := stop time - start time
end for
```

```
computation time := (compute coeff) * (warmup time/(Max warm up runs))

/* test */
start overall time
for i:=0 to (Max testbed runs)
 start time
 call collective
 stop time
 collective time := stop time - start time

 start time
 compute for 'computation time' amount
 stop time
 computation time := stop time - start time
end for
stop overall time

/* measure */
overall time := stop overall time - start overall time
collective time := collective time/(Max testbed runs)
computation time := computation time/(Max testbed runs)

/* report */
if(MYTHREAD = 0) then
  find max collective time for all threads,
    and record the slowest thread
  report collective time, computation time, overall time
   for the slowest thread
end if

end procedure upc_mtu_testbed()
```

The collective time, computation time and the overall time (time from the start of the test to the end) are the times obtained from the test runs, over all threads. These values are stored in a shared array, and eventually `THREAD 0` averages the collective time for all threads, compares for the largest average, and prints this as the collective time. It also prints the overall and the averaged computation time for this thread (the slowest thread), along with other data values. The computation time, collective to be tested, startup skew etc are specified during runtime, while computational skew, synchronization mode etc are specified during compilation.

7.2 Testmatrix and testcases

Upon completion of our library and benchmarking application development, we created a sets of comparitive *test cases* to compare different libraries and implementations over varying run-time systems for different collectives etc. This information is presented in the *GMTU test matrix* (below).

GMTU Test Matrix

	Collective	mPULL	mPUSH	GMTU_PUSH	BERKELEY_mPULL	BERKELEY_mPUSH	Computation Time	Fair	IN_SYNC	OUT_SYNC
Testcase 0	Broadcast Exchange	X	X	Mem Reg	X	X	small	yes	ALL	ALL
Testcase1	ALL	X	X	Mem Reg	X	X	Coll x 2	yes	ALL	ALL
Testcase2	ALL	X	X	Dyn Alloc	X	X	Coll x 2	yes	ALL	ALL
Testcase3	Broadcast Exchange	X	X	Mem Reg	X	X	Coll x 2	skew	ALL	ALL
Testcase4	Broadcast	X		Mem Reg	X		Coll x 2	yes	ALL	ALL
Testcase5	Broadcast		X	Mem Reg		X	Coll x 2	yes	ALL	ALL
Testcase6	ALL	X	X	Static Alloc	X	X	Coll x 2	yes	ALL	ALL
Testcase7	Broadcast Exchange	X	X	Mem Reg	X	X	Coll x 2	yes	MY	MY
Testcase8	Broadcast Exchange	X	X	Mem Reg	X	X	Coll x 2	yes	NO	NO
Testcase9	Broadcast Exchange	X	X	Dyn Alloc	X	X	Coll x 2	yes	MY	MY
Testcase10	Broadcast Exchange	X	X	Dyn Alloc	X	X	Coll x 2	yes	NO	NO
Testcase11	Broadcast Exchange	X	X	Static Alloc	X	X	Coll x 2	yes	MY	MY
Testcase12	Broadcast Exchange	X	X	Static Alloc	X	X	Coll x 2	yes	NO	NO

Figure 21. GMTU test matrix

The testcases one, two and six compare the three different memory allocation schemes in the GMTU library, respectively, with the reference implementation, over the MuPC runtime system (`MuPC-mPULL` and `MuPc-mPUSH`) and the GASNet runtime system (`Berk_mPULL` and `Berk_mPUSH`). These tests were conducted for all the collectives and using `IN_ALL_SYNC` and `OUT_ALL_SYNC` modes throughout, and with uniform computation times among all threads, computing for approximately twice the collective communication time - as measured during warm-up runs. We conduct the same comparison under different combinations of synchronization modes over test cases seven through tweleve. These test are conducted for broadcast and exchange collectives only, because the 'one to all' and 'all to all' are the two extreme communication patterns.

The MuPC runtime system as mentioned earlier uses a cache, unlike Bekeley's runtime system. The line length of the cache is 1K and there are 256 lines in the cache. The cache impacts MuPC's performance, as depicted in the naive push and pull tests - test case four and five, respectively. These testcases only compare the `upc_all_broadcast` collective, from the naive implementation versus the GMTU library, so as to highlight the need for collective communication operations against a user-level array assignment based implementation.

In testcase three, we compare the `upc_all_broadcast` and `upc_all_exchange` collective implementations only to show the impact of unbalanced computational load over different processes, for the two extreme communication instances. These tests compare results from a *normal* test run, where computation time is nearly twice that of collective time during warm-up and uniform over all threads, against an *unbalanced* test run where one thread computes ten times longer than the other threads.

The results for the tests, outlined in the test matrix, along with our observations are provided in the following pages.

7.3 Naive push and pull based comparisons

- upc_all_broadcast naive push and pull
  A test of the naive broadcast for two threads amounts to a measurement of the point-to-point communication time because the only remote operation consists of the source thread sending a single message to the (sole) destination thread. Implemented using array assignments, for n elements in the source thread, this operation amounts to as many sends from the source in the push based implementation. The situation grows worse in the pull implementation, as observed in the data, since a request by the destination thread (for a pull) preceeds a send by the source, for every array element!

  These data therefore reflect the need for specific collective communication operations that would translate to fewer remote operations (like upc_memcpy() based reference implementation) for performing relocalization operations.

  The results also show that the MuPC runtime system with a 1K cache has a performance advantage over Berkeley's run time system, which currently does not have a cache.

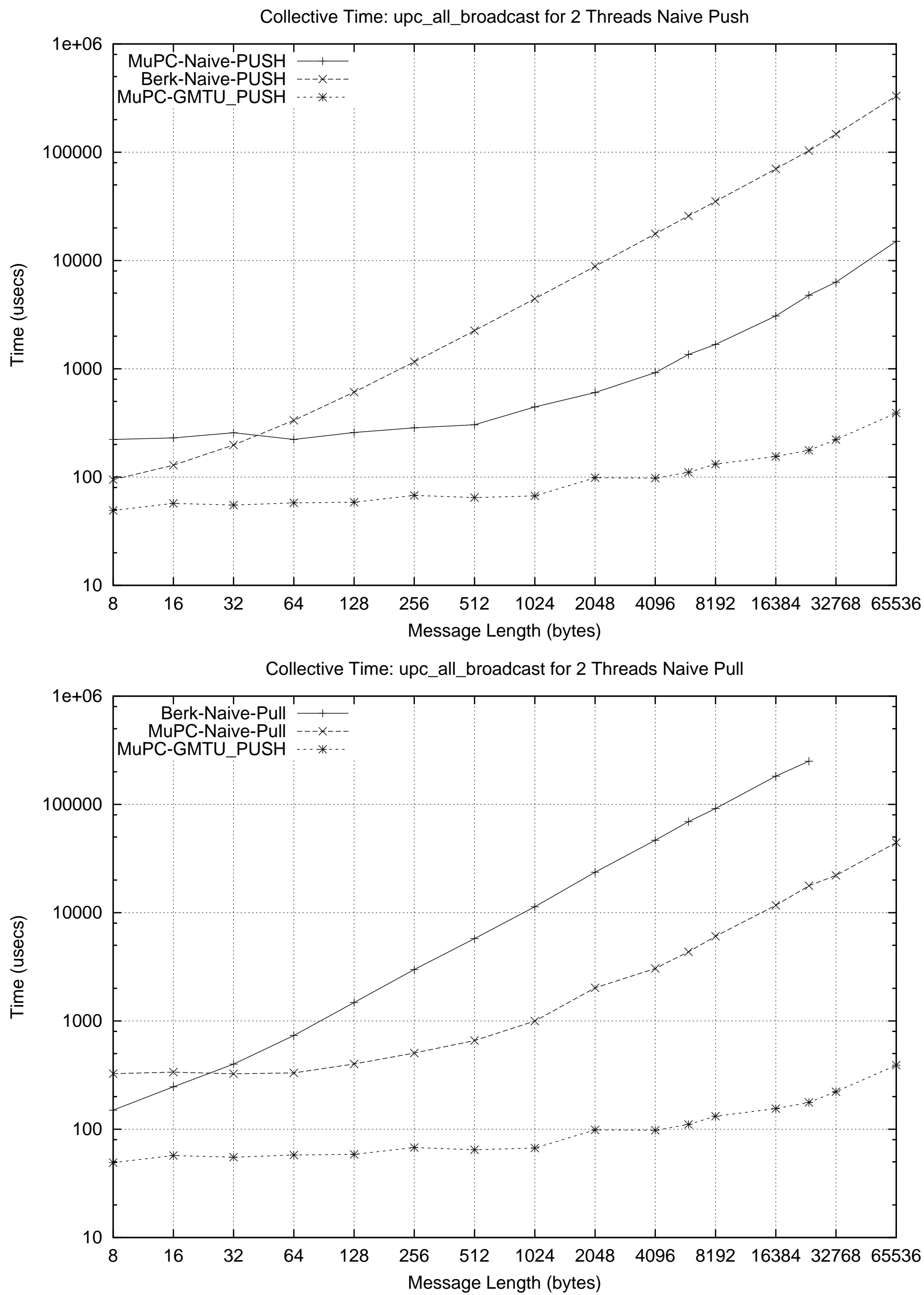


Figure 22. Naive push/pull vs reference implementation, MuPC and GASNet, and GMTU memory registration based push

7.3.1 Comparison of GMTU algorithms

The GMTU library consists of various implementations of UPC collective communication operations, depending on the memory allocation scheme choosen. The results below highlight the performance differences between these implementations.

- `upc_all_broadcast`
  The performance of broadcast for 16 threads, with memory registration compared with static and dynamic memory allocation, using the GMTU library (compiled under MuPC) are shown in Figure 23). All the algorithms are implemented using the spanning tree based broadcast and use the token based send/receives (even in the memory registration scheme), for message passing. The data shows that the performance of all the three algorithms are comparable for small message lengths, and for large message lengths the static buffer algorithm performs slightly better than the memory registration or dynamic buffer algorithms.

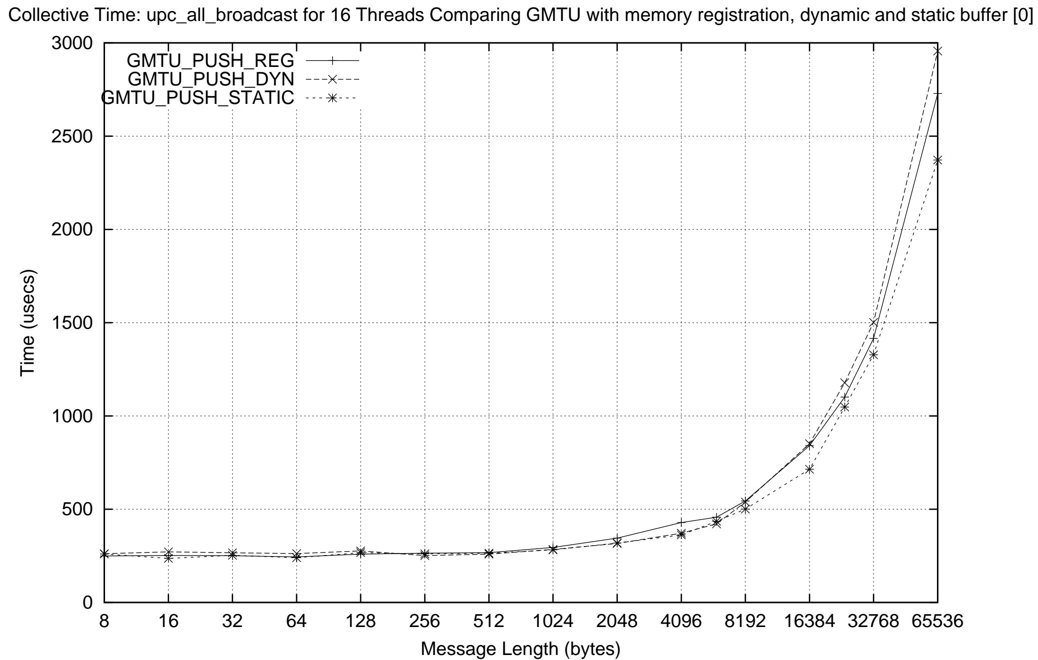


Figure 23. `upc_all_broadcast` **for 16 threads with GMTU memory registration, dynamic and static allocation**

- `upc_all_scatter`

  For scatter(Figure 24), the memory registration based algorithm is implemented as a straight push using remote puts, while the dynamic buffer based algorithm uses the spanning tree based push. The static buffer based algorithm also uses the spanning tree based push, but only until the buffer size is larger (or equal to) `nbytes * THREADS/2` In these results, we have a static buffer of size 8k and for 16 threads the buffer is large enough to send messages upto 1K - beyond this limit the static buffer algorithm uses the straight push for message lengths upto 8K. Beyond the 8K limit, the message is broken up into chunks of 8K and sent using the straight push algorithm. The results show the effect of this buffer limit, as the performance of the static buffer algorithm degrades after message length of 1K.

  The tree based dynamic push performs worse than the memory registration based push because of additional message that is sent along the tree, this is because unlike broadcast, where the message length is fixed, in scatter the largest message sent to a receiver is `nbytes * THREADS/2` during the first round, and the smallest is `nbytes` during the last round. The overhead of sending additional data is represented in the results.

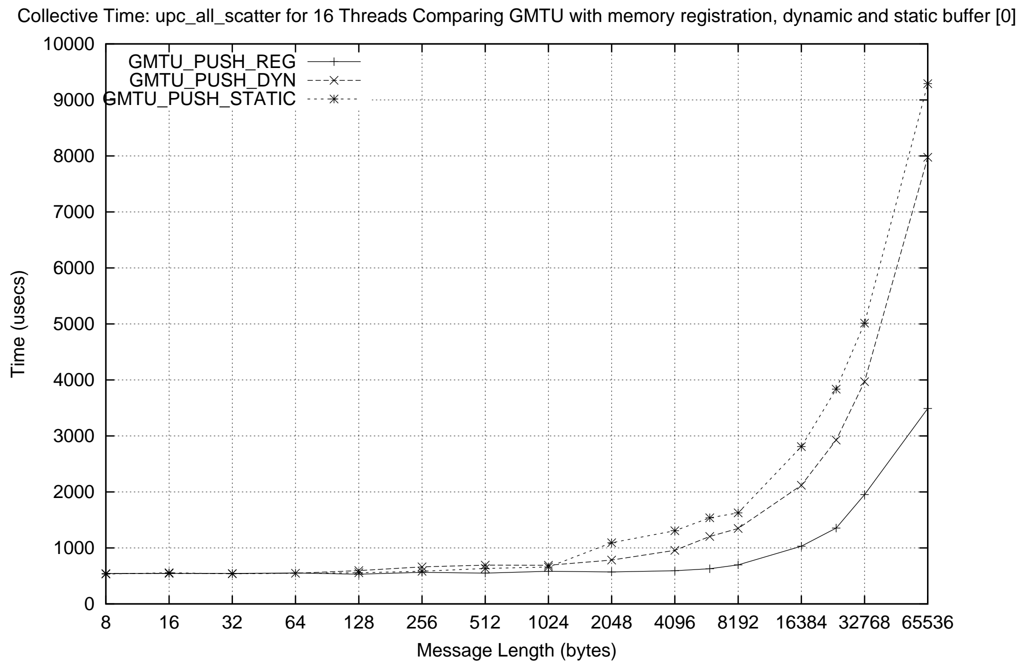


Figure 24. `upc_all_scatter` **for 16 threads with GMTU memory registration, dynamic and static allocation**

- `upc_all_gather`

  The gather collective (Figure 25) is implemented using remote puts in the memory registration algorithm, where all the `THREADS-1` source threads issue a remote write directly into the destination thread's shared destination array. On the other hand, the dynamic buffer and static buffer algorithms are implemented using a polling based receive, where for the static buffer case additional synchronization messages are issued among source threads to indicate the destination buffer is available. The process of polling requires releasing receive tokens and adds an extra overhead at the destination thread. In the dynamic allocation environment this cost increases with the message length, as the size of the buffer allocated and the cost of copying the message from the buffer, into the destination array, increase. In the static buffer algorithm, the cost of polling for the message and the cost of an extra (short synchronization message) are the additional overheads. The memory registration scheme therefore performs better due to lack of any additional overhead as discussed above.

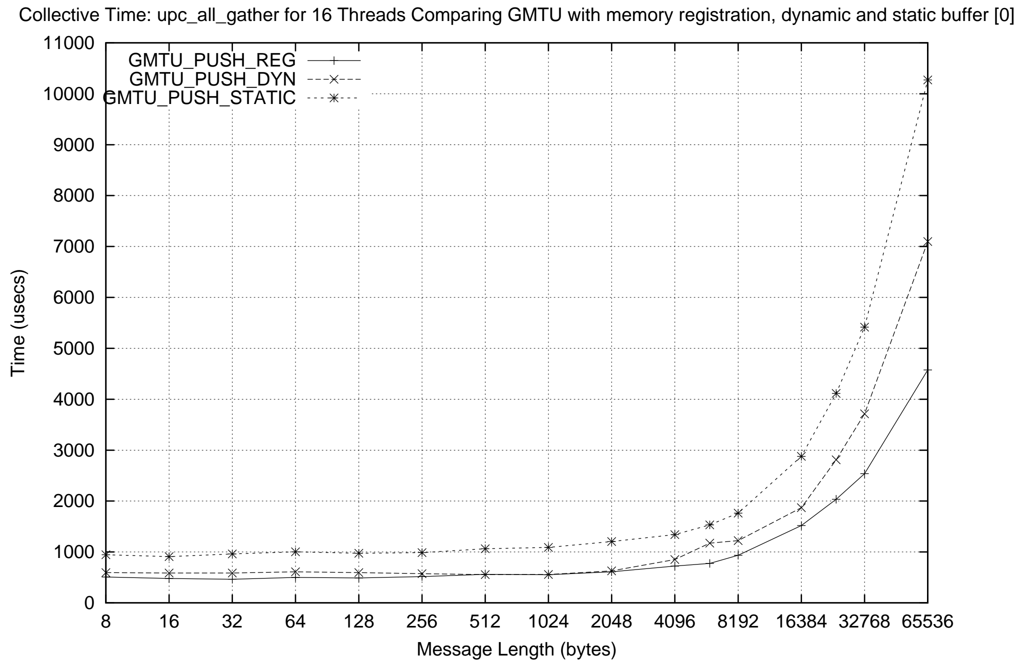


Figure 25. `upc_all_gather` **for 16 threads with GMTU memory registration, dynamic and static allocation**

- `upc_all_gather_all`

  In the 'gather all' collective, the communication pattern is similar to all threads performing a broadcast of `nbytes` of data to all the other threads(Figure 26). The memory

registration based algorithm uses remote puts, where threads 'put' the data directly into the shared memory region of their partner threads (using the XOR based algorithm). The dynamic and static buffer based algorithms use a ring based send/receive scheme where each thread sends to its 'next' thread and receives from its 'previous' thread. We observe that the remote put based algorithm is the fastest, as there is no extra overhead from copying the message (from the receive buffer into the shared destination array). The data also show that the performance of the dynamic and static buffer algorithms are similar until the buffer size is reached in the static buffer algorithm, and therefore beyond the 8K mark the packetization by our application adds an additional overhead.

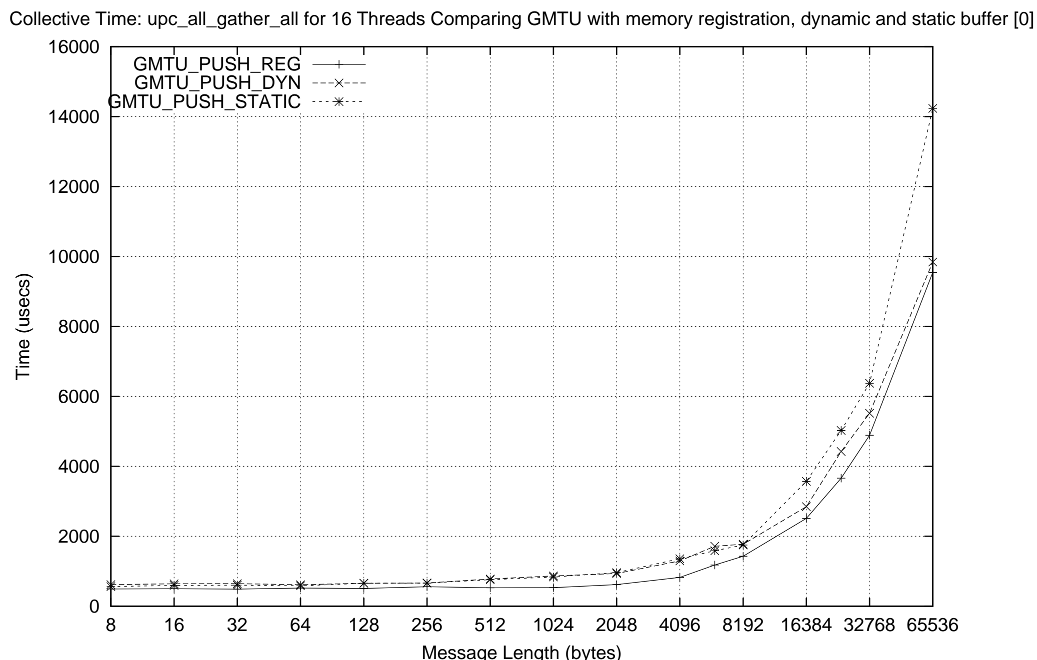


Figure 26. `upc_all_gather_all` **for 16 threads with GMTU memory registration, dynamic and static allocation**

- `upc_all_exchange`
  The exchange collective (Figure 27), using the memory registration algorithm is implemented as a remote put based scatter from all threads, where each thread writes to a remote buffer (shared destination array) on its partner thread (partner calculated using the 'XOR' process). While the static buffer based exchange implementation uses the same algorithm, the lack of a 'tag' based send in GM implied that extra synchronization messages were required to send/receive data. On the contrary, the dynamica allocation algorithm uses the purely polling based scheme where all threads allocate a large enough

receive buffer and first free all the send tokens, then send their data to all the other threads and poll for messages from them. This implementation, as observed from the results, provides no additional performance improvement over the static buffer algorithm with the extra message overhead, and both these algorithms perform worse than the memory registration based algorithm due to overhead from copying the data into the destination buffer and polling for messages.

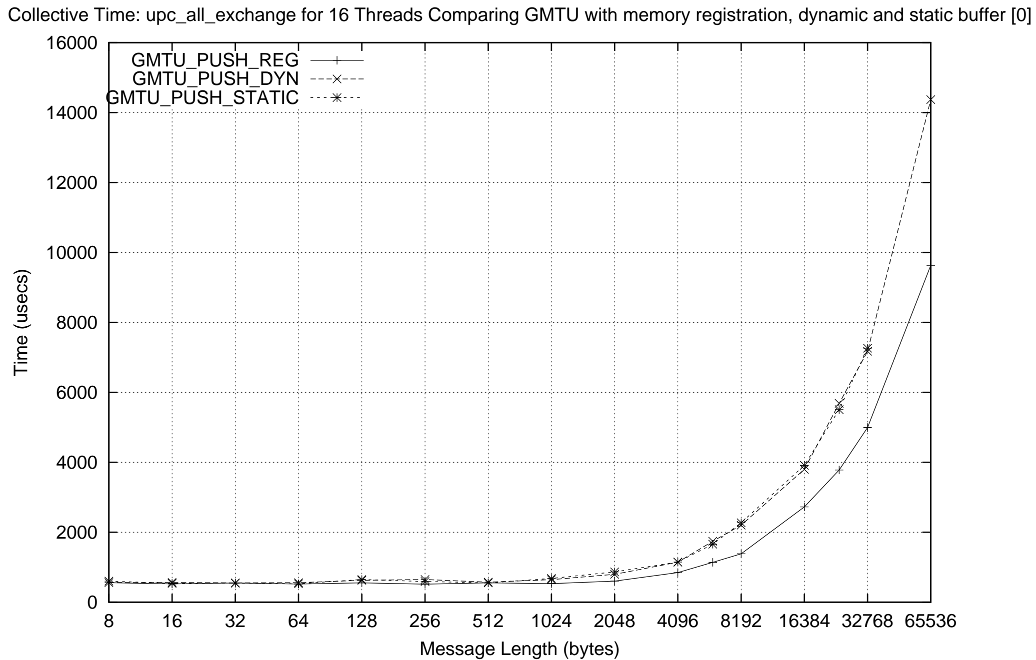


Figure 27. `upc_all_exchange` **for 16 threads with GMTU memory registration, dynamic and static allocation**

- `upc_all_permute`
  The permute collective (Figure 28) amounts to a point-to-point send as all nodes attempt to send data to their respective destinations, which are distinct for all threads. Our tests consisted of a permute vector where each thread sent its data to its 'next' thread. Therefore, in the memory registration and remote put based algorithm, threads simply wrote the data onto the next thread's shared destination array and waited at the next synchronization point. In the dynamic and static allocation schemes, using the regular send and receives, threads first send their data to their next threads and receive from their previous thread. This also, co-incidentally, is the first step of the ring based 'gather all' collective, where each thread sends `nbytes` to its next thread. The overhead of polling for a message, and copying the data from the receive buffer into the destination array has an

impact on the performance of the dynamic and static algorithms. Also, when the message length is larger than the buffer size, for the static buffer algorithm's test case, we observe additional overhead due to packetization as described earlier.

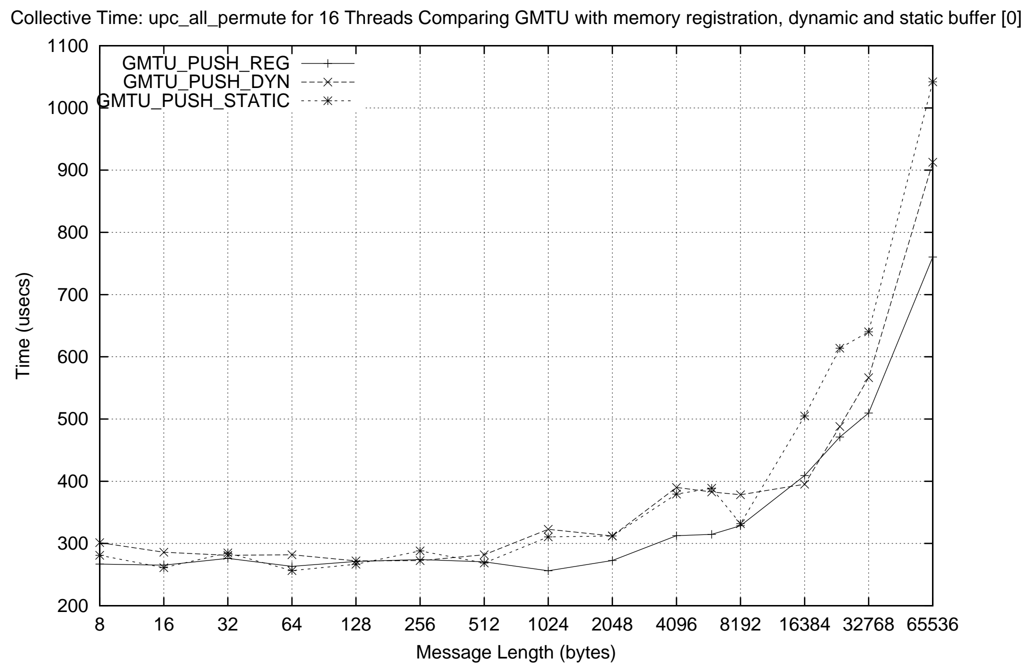


Figure 28. `upc_all_permute` **for 16 threads with GMTU memory registration, dynamic and static allocation**

These data indicate that the memory registration based algorithm is consistently better than the other two algorithms, and we shall therefore only compare this implementation with the reference implementation for our collective comparisons, in the next subsection. The data also point to the impact of the static buffer size on the overall performance of the algorithm.

7.4 Comparison of collective communication libraries

The results from comparing the GMTU library based functions indicate that the memory registration based algorithms perform the best. Therefore, in the following comparisons between the reference implementation based collective, the GMTU implementation uses the memory registration scheme.

The comparisons were performed for fairly high computation times, which were uniform across all threads and were approximately twice the collective time mesasured during warm-up. These test always use the `IN_ALL_SYNC` and `OUT_ALL_SYNC` modes, and the results for 2 and 16 threads comparing the GMTU algorithm and the reference implementation are shown. Also, each comparison is made for the collective compiled under the same UPC compiler and executed under the same runtime system. Therefore, for each collective we have 4 graphs, two indicating the comparison under MuPC runtime system for 2 and 16 threads and two more for Berkeley's UPC and GASNet runtime system.

The results indicate in general over all collective and for 2 threads, the 'push' based reference implementation over the Berkeley runtime system always performs better than the 'pull' or GMTU push based collectives. We also observe that the cost of push based reference implementation, over the MuPC runtime system, for very small message lengths (less than and equal to 16 bytes) is unequal and higher than the cost of sending a slightly larger message. Also, this is never the case in Berkeley's run time system's results, using the same reference implementation.

These results point to the fact that for the small message push, where message length is less than or equal to 16 bytes, a remote operation in the MuPC runtime system does not occur until the cache is filled or a synchronization event occurs. It is, therefore the additional cost of waiting for the cache to be invalidated that is reflected in the results.

The results for the MuPC based comparisons always show a greater amount of improvement, as compared to the Berkeley UPC based comparison, because of the additional latency cost in the MuPC runtime system imposed by the MPICH-GM layer. The performance of Berkeley's UPC based reference implementation is comparable to the straight push based GMTU implementations (for example in scatter) as they both operate on the same messaging layer.

However, there are some differences in the performance of the GMTU push based implementations and Berkeley's UPC based reference implementation, for varying thread ranges (example: push based reference implementation is faster for 2 threads but slower for 16 threads, than GMTU push based implementation). This could be due to additional latency in preparing the message at the application level (GASNet or GMTU), and due to the type of implementation used. Therefore, in most cases where we use collective specific optimization in the GMTU library, we find our performance improves against the reference implementation as the thread

size increases. This observation is made clearer by looking at comparions over 2 threads and then over 16 threads, and therefore we show only these two comparisons to highlight any improvements in our algorithm.

The following subsections compare the performance of our implementation using the memory registration scheme, over 2 and 16 threads, against the reference implementation over 2 different compilers/run time systems.

7.4.1 `upc_all_broadcast`

The comparison over 2 threads amounts to a test of point-to-point messaging performance, where one thread sends to the next. In the pull based reference implementation (MuPC)this cost is higher because an additional message, sent by the destination thread, is required to 'request' the source to send it the message. Therefore the push based times are better for 2 threads. The GMTU push is a not a remote put in the case of broadcast, rather the source send the data to the destination - which polls after allocating a buffer. The buffer allocated is the registered destination array itself, and therefore the receving thread does not have to re-copy the data.

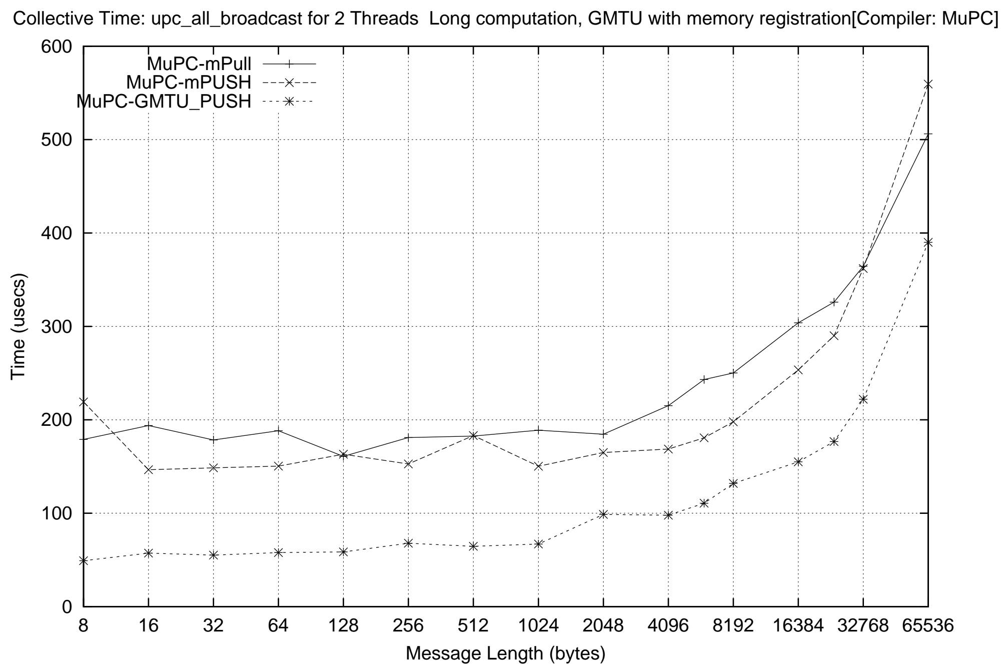


Figure 29. `upc_all_broadcast` **for 2 threads over MuPC**

Over 16 threads, the pull based reference implementation (MuPC) performs much better than

the push implementation. This is because no sequentiallity is imposed in the pull implementation, and ready threads can message the source and receive the data right away. The GMTU algorithm performs a lot better due to the spanning tree based implementation that only takes `log(THREADS)` steps.

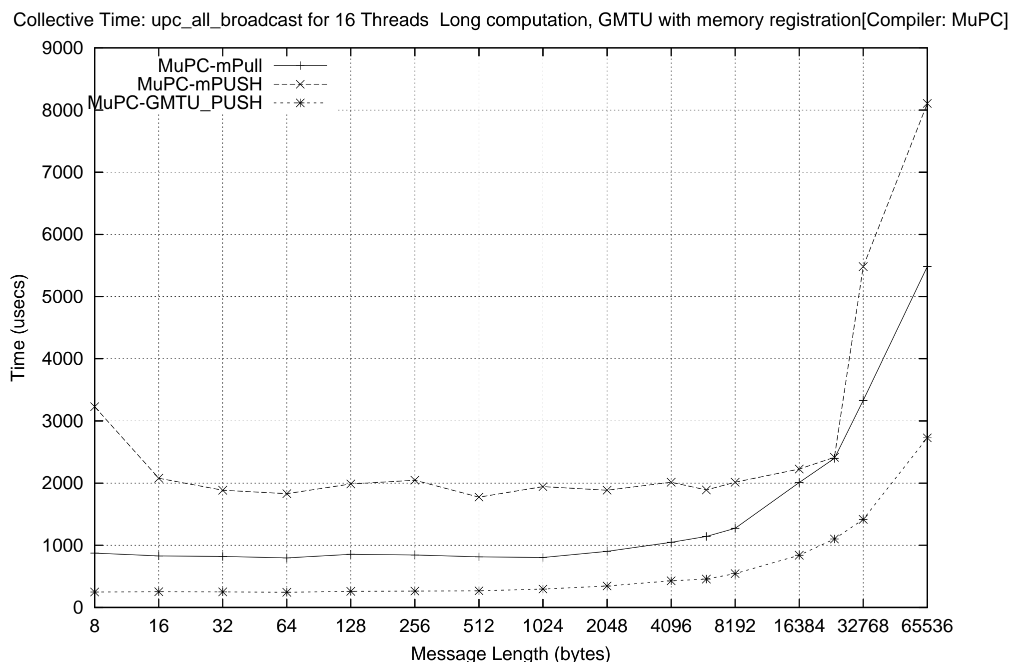


Figure 30. `upc_all_broadcast` **for 16 threads over MuPC**

Over Berkeley's UPC runtime system, the reference implementation push based algorithm's performance should be comparable to the GMTU push based performance as they both amount to the same operation, over the same messaging layer. Once again we observe that the pull based reference implementation is slower than the push based implementation due to extra messaging.

For 16 threads, the spanning tree based GMTU algorithm scales well and performs a lot better than the push and pull based reference implementation, despite the presence of the same messaging layer. This shows that our improvments over the reference implementation are also compiler independent.

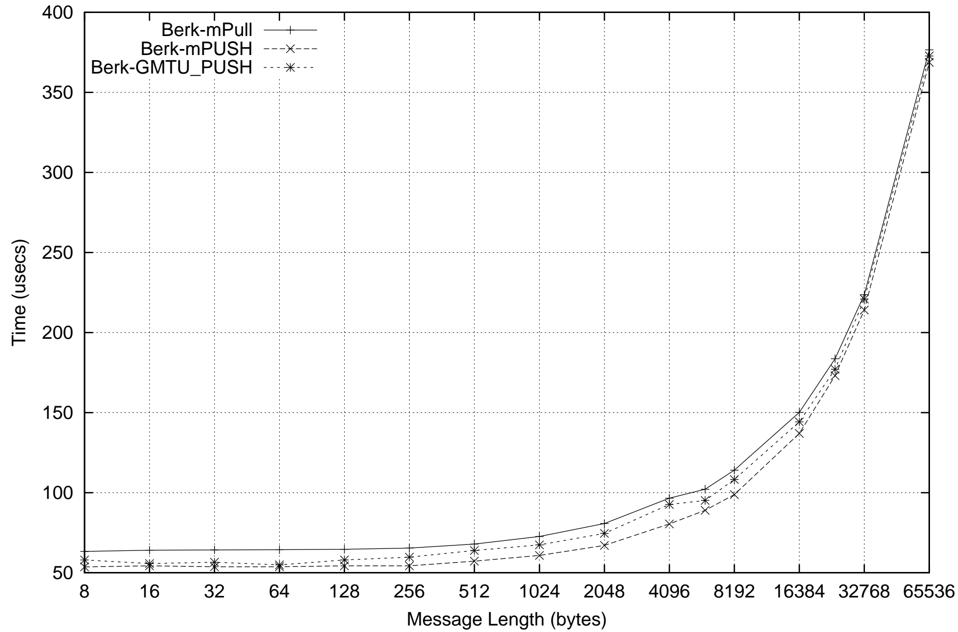


Figure 31. `upc_all_broadcast` **for 2 threads over GASNet runtime system**

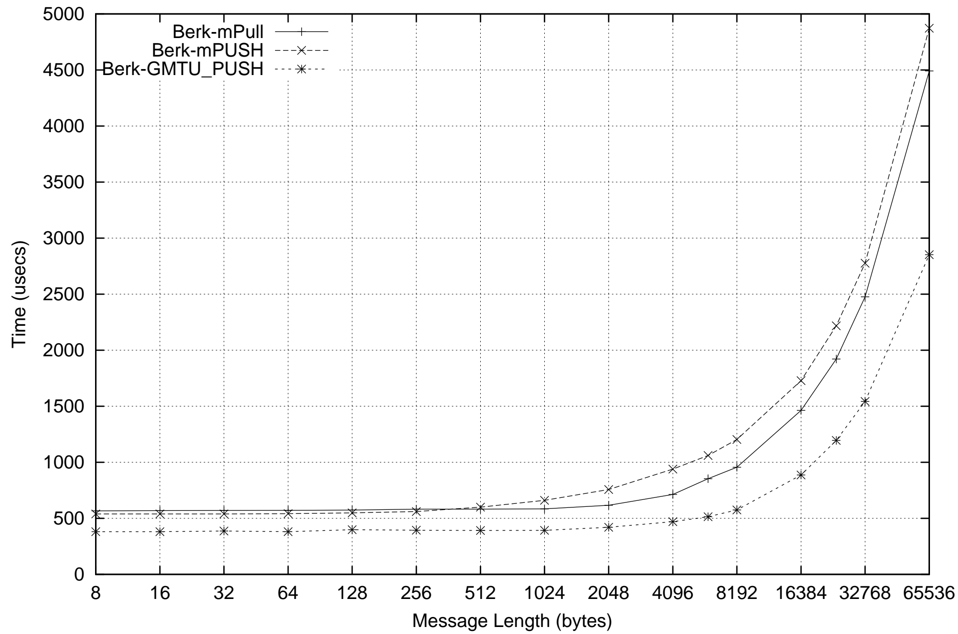


Figure 32. `upc_all_broadcast` **for 16 threads over GASNet runtime system**

7.4.2 `upc_all_scatter`

The push based reference implementation of scatter also performs better than the pull based implementation for 2 threads. The results for the GMTU based push based scatter, for 2 threads, show the time it takes for the source thread to perform 1 remote put of `nbytes`. When this is compared with the result for 2 threads in broadcast, it amounts to comparing regular send/receives against remote puts in the GM message passing system, with both using memory registration. According to the GM documentation [**?**], the remote put operations are on average 10% slower that the token based send/receive functions. Therefore this cost, and the cost of extra synchronization in the scatter algorithm, due to its data distribution pattern, are seen when comparing the performance of broadcast and scatter collectives (for GMTU).

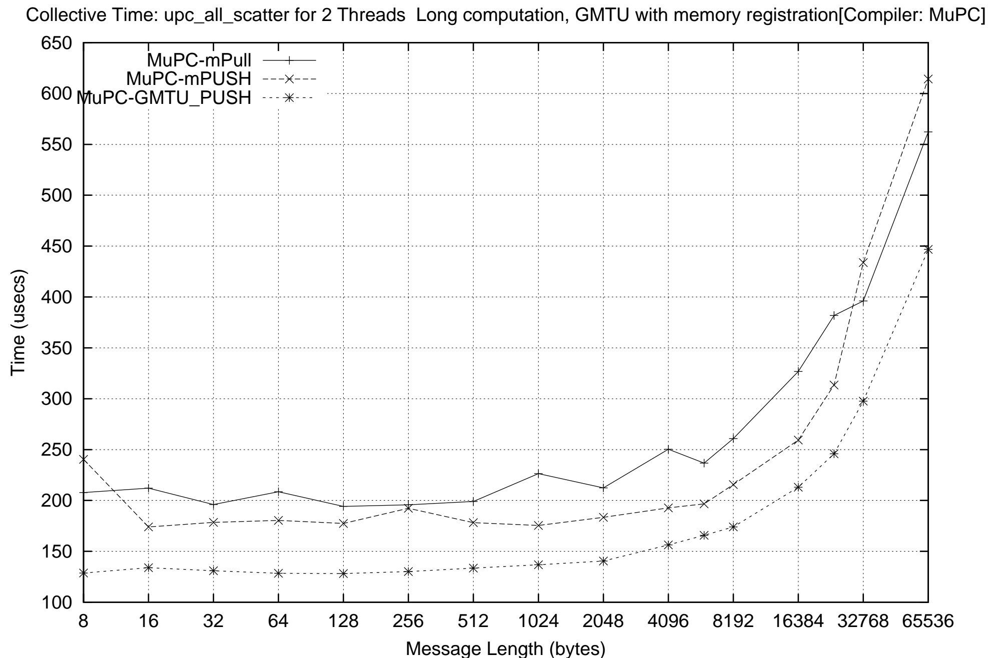


Figure 33. `upc_all_scatter` **for 2 threads over MuPC**

Over 16 threads, the pull based reference implementation, being more parallelized, performs better than the push based implementation. However, the memory registration based straight push scatter in GMTU implementation performs far better despite being a straightforward implementation (not spanning tree is used either). The improvements are due to the comparative advantage of directly using GM, and due to the memory registration based implementation of the algorithm. This is more apparent when we view the performance of the GMTU collectives against the reference implementation in the Berkeley runtime system.

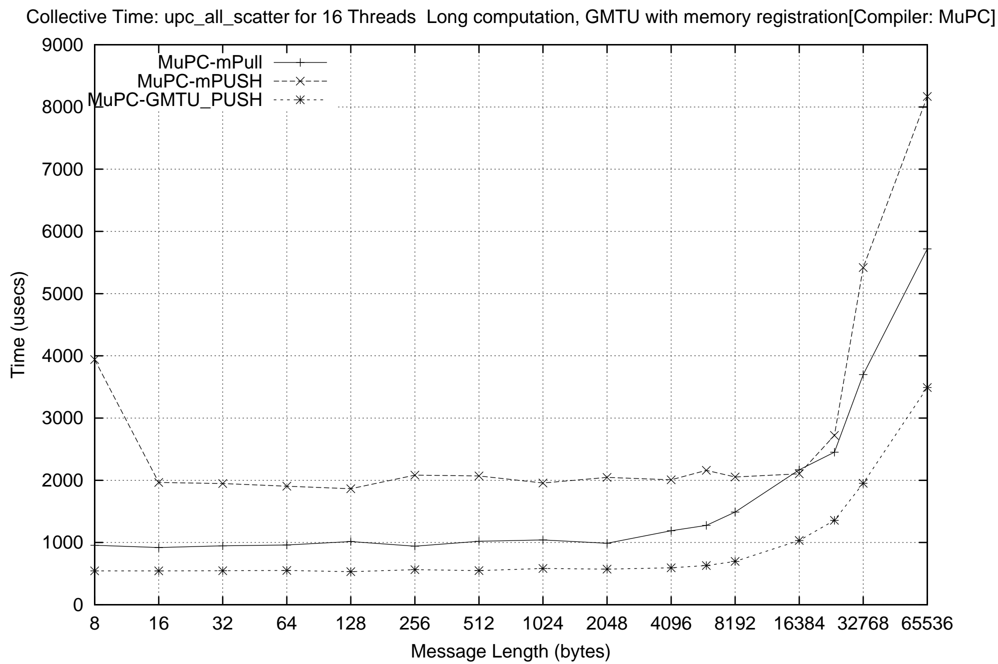


Figure 34. `upc_all_scatter` **for 16 threads over MuPC**

Using the Berkeley UPC compiler and runtime system we compared the reference implementation based scatter against the GMTU memory registration based scatter and observe that the performance of the pull based reference implementation improves with the thread size for large messages. However, the GMTU implementation improves by a much larger margin for large messages due to the absence of any mem-copying overhead.

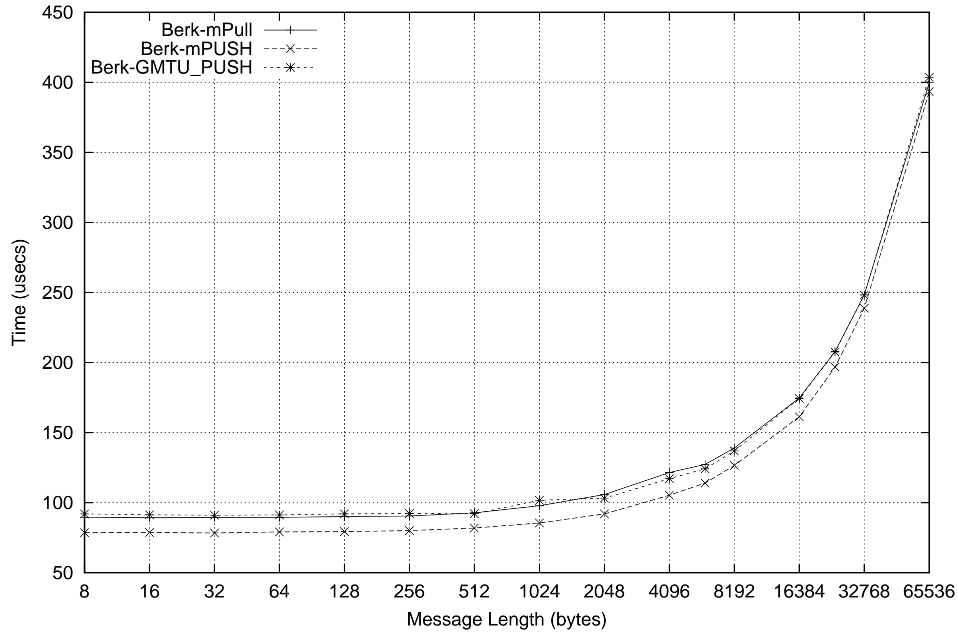


Figure 35. `upc_all_scatter` **for 2 threads over GASNet runtime system**

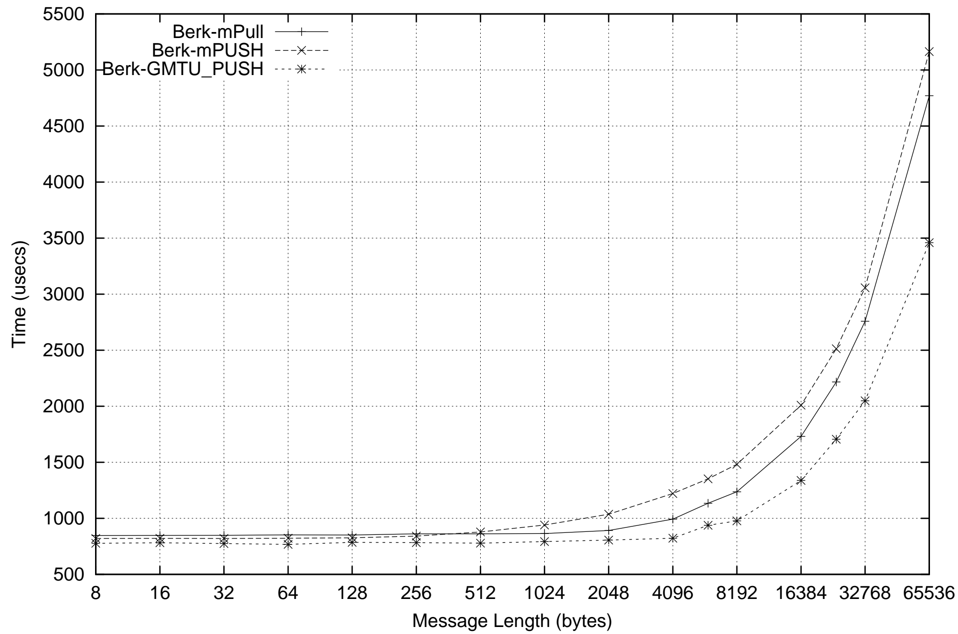


Figure 36. `upc_all_scatter` **for 16 threads over GASNet runtime system**

7.4.3 `upc_all_gather`

The push and pull based gather collectives from the reference implementation, over the MuPC runtime system, display the effect of the 'cache invalidation' cost, for message lengths less than 16 bytes as discussed earlier. A push based gather amounts to one remote put from the source, over 2 threads, and in the pull based implementation, the same process occurs with an additional message from the destination to the source requesting the remote put. Therefore, in both cases we see the cost of the cache invalidation in the cache based MuPC runtime system. The memory registration based GMTU implementation, amounts to a remote put by the source thread and is similar to the push based reference implementation of the gather collective.

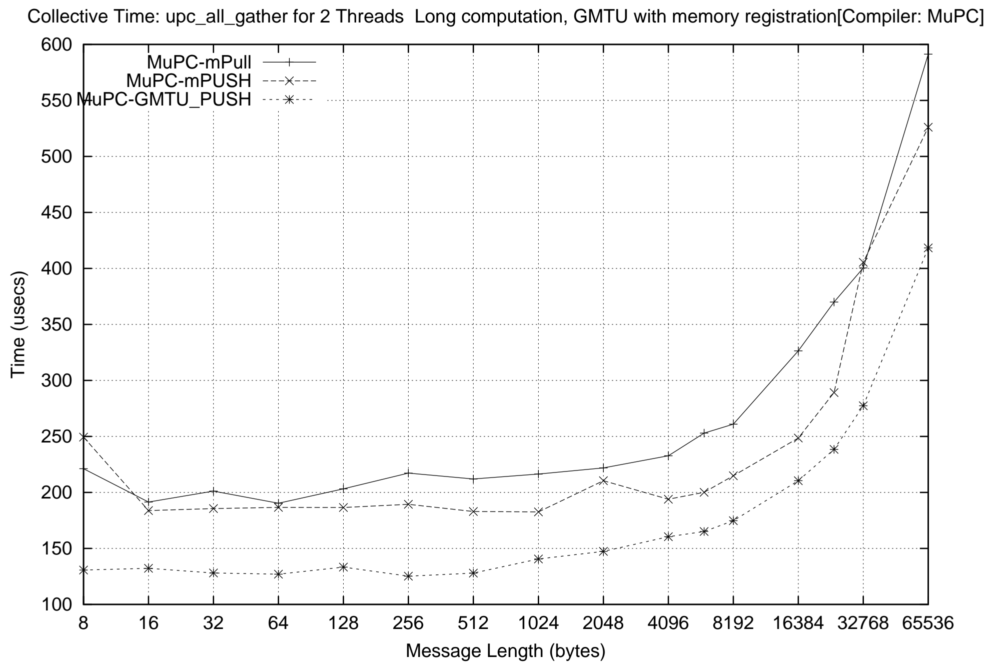


Figure 37. `upc_all_gather` **for 2 threads over MuPC**

Over 16 threads, we observe that the performance trend of the push based reference implementation of gather and that of the GMTU push based gather are similar. The GMTU algorithm performance is better as it uses a zero-copy remote write using GM's messaging layer.

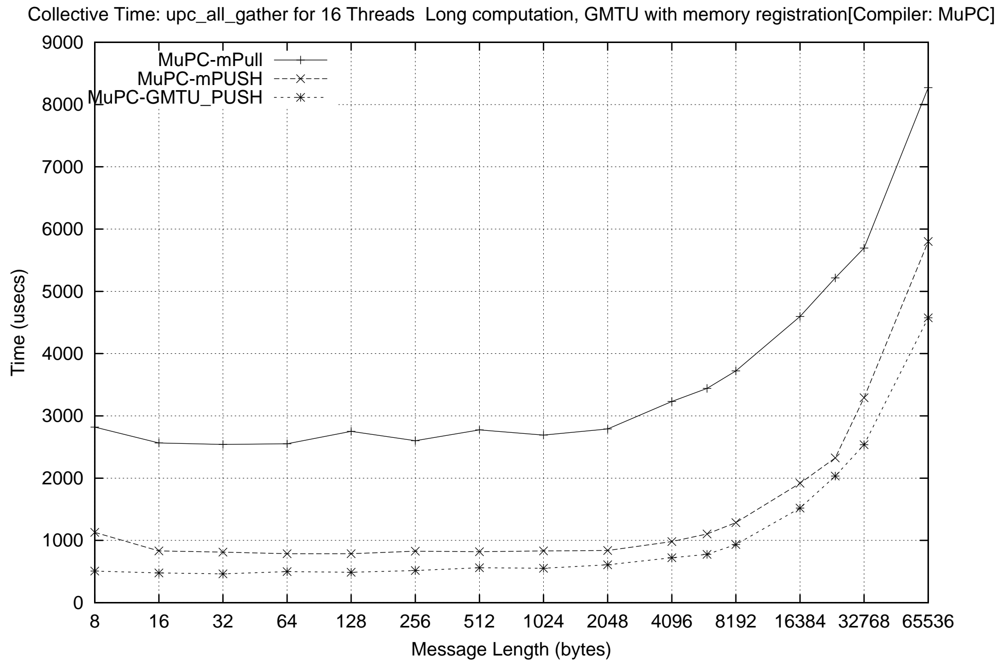


Figure 38. `upc_all_gather` **for 16 threads over MuPC**

The comparison of the gather collectives over the Berkeley runtime system show that the GMTU implementation's performance is comparable to the push based reference implementation, over 16 threads. However, when we observe the graphs for 2 and 16 threads, it becomes apparent that the GMTU remote put based implementation scales better than the push based reference implementation, and although their performance times are similar at 16 threads we can expect the GMTU algorithm to perform better for larger thread size.

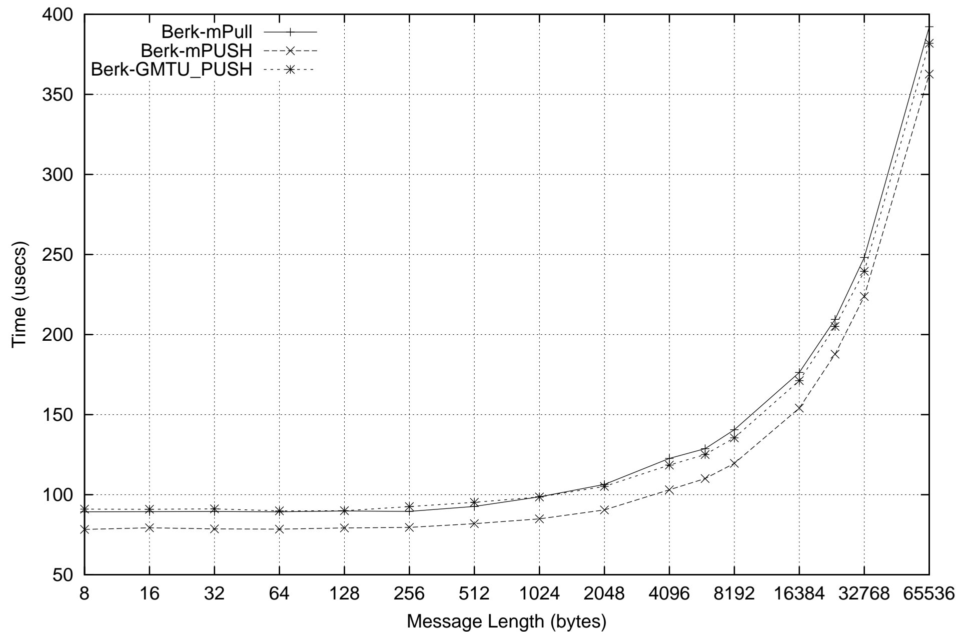


Figure 39. `upc_all_gather` **for 2 threads over GASNet runtime system**

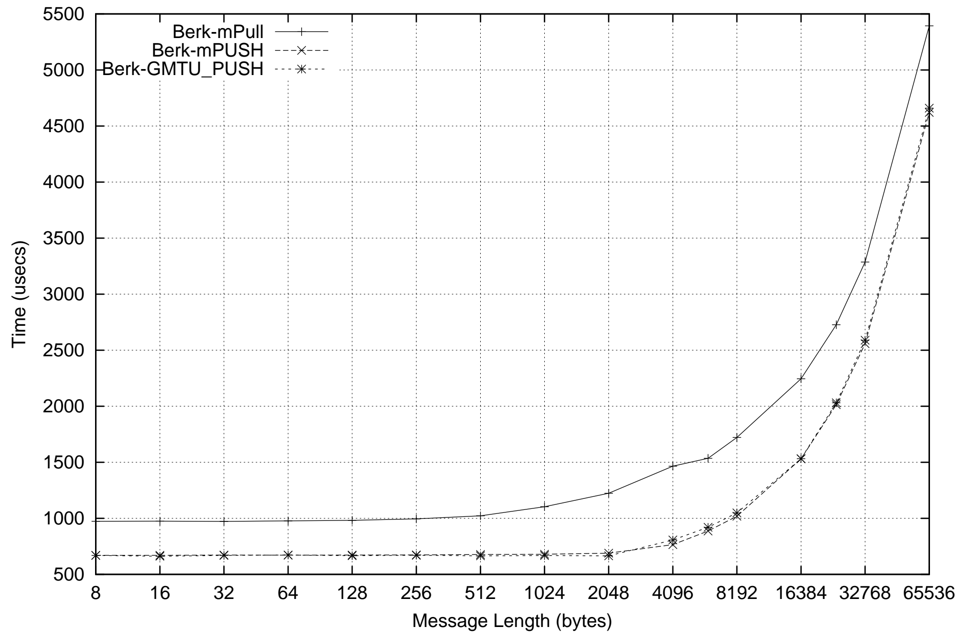


Figure 40. `upc_all_gather` **for 16 threads over GASNet runtime system**

7.4.4 `upc_all_gather_all`

The GMTU memory registration based gather all collective, implemented as a `THREADS-1` step broadcast by each thread and for 2 threads amounts to two remote puts by each thread. In the MuPC runtime system, the push based reference implementation, for 2 threads, is similar to each thread issuing a remote put. The pull based implementation is similar, but involves extra messages for requesting a remote put.

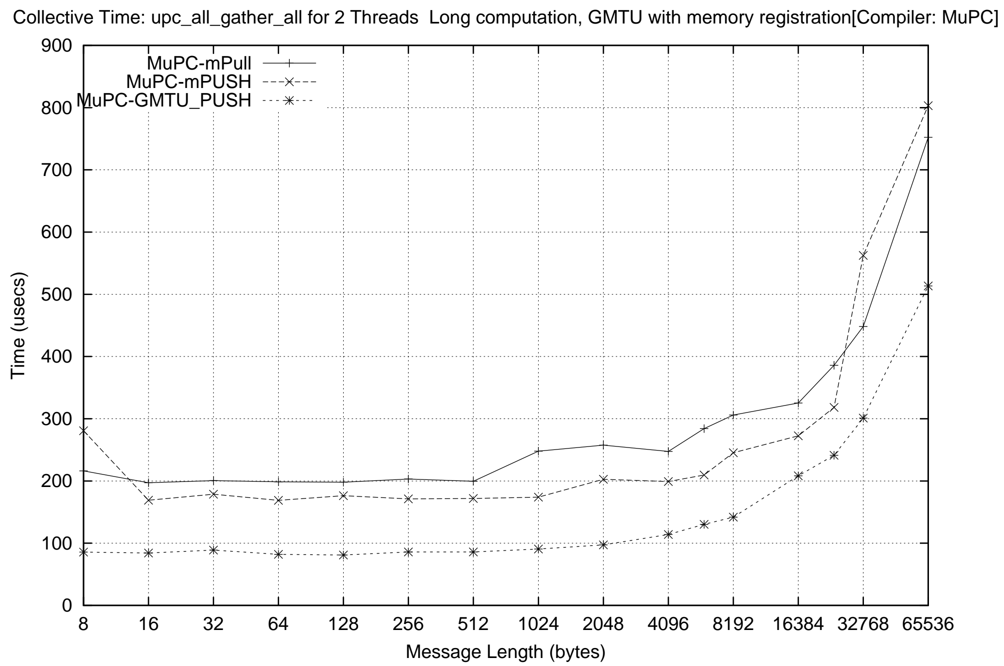


Figure 41. `upc_all_gather_all` **for 2 threads over MuPC**

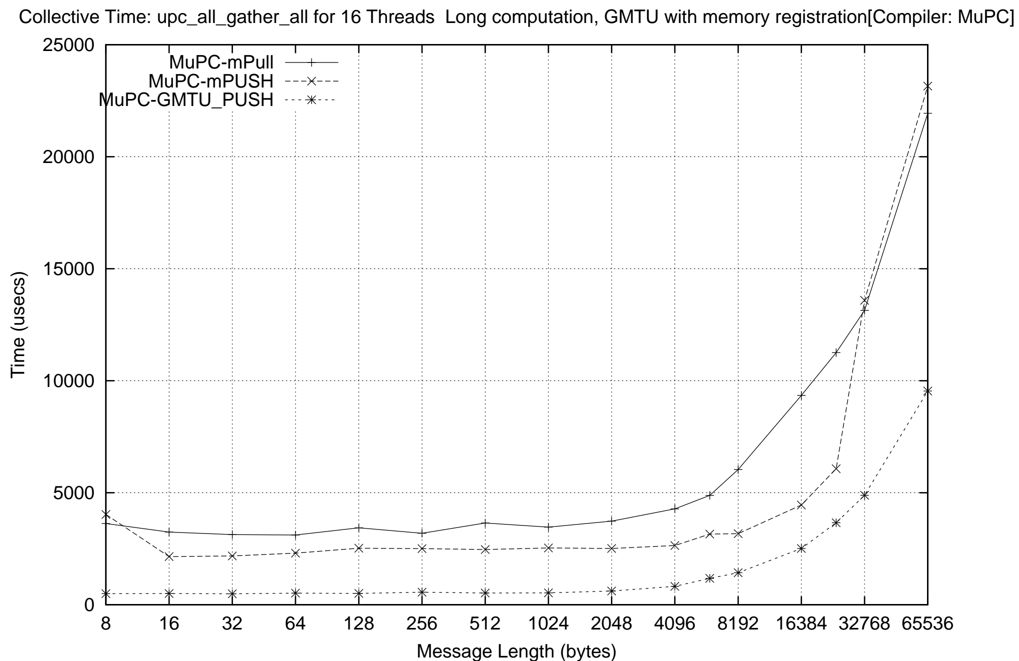


Figure 42. `upc_all_gather_all` **for 16 threads over MuPC**

Over 16 threads, the push based reference implementation performs better than the pull based gather all implementation because all threads send the same amount of data in this collective. The advantage due to parallelization using the pull in the 'one-to-all' collectives is not longer a factor in 'all-to-all' collectives such as this. The GMTU push based algorithm performs the best overall, in the MuPC runtime system due to the remote puts and its zero-copy performance.

In the comparisons over Berkeley's UPC runtime system, the push based reference implementation performed the best over 2 threads, and compared to the pull based implementation it performs better over increasing thread sizes. The GMTU remote put based algorithm scales well and performs better than the reference implementation over 16 threads. This difference is more pronounced for larger message lengths due to reduced copying costs in the GMTU algorithm, where the message arrives directly into the, registered, shared destination array.

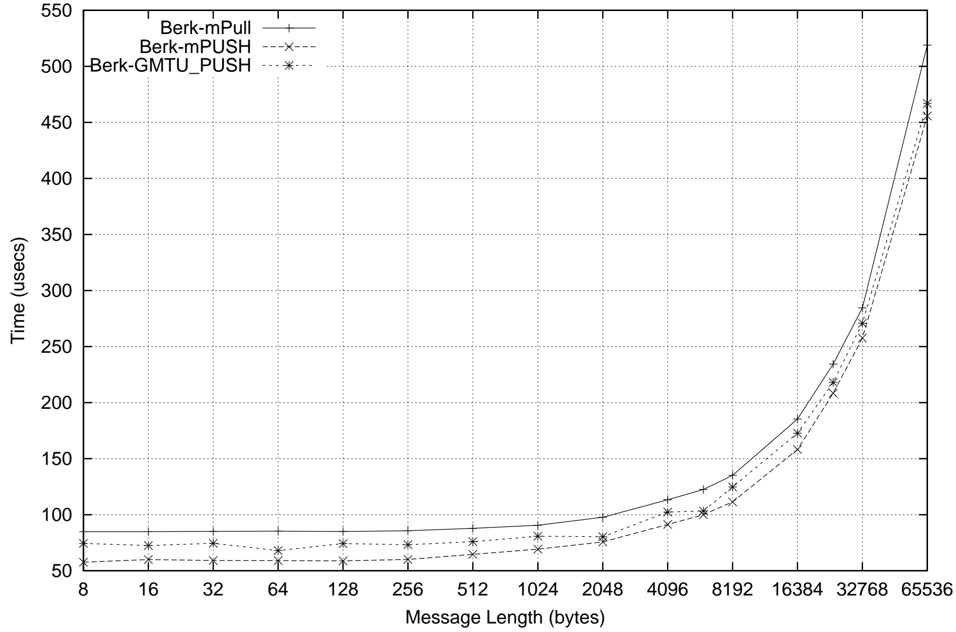


Figure 43. `upc_all_gather_all` **for 2 threads over GASNet runtime system**

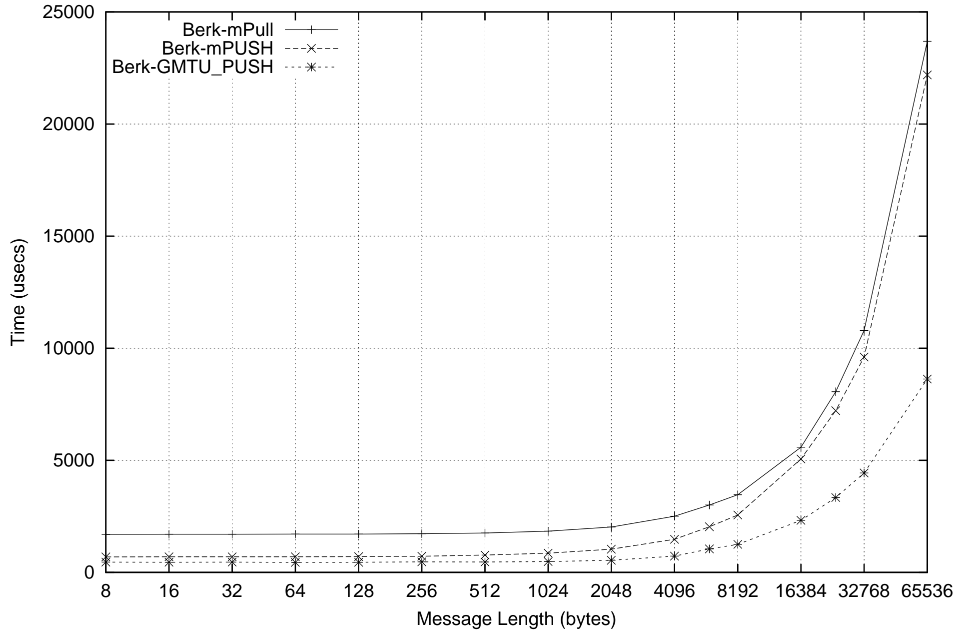


Figure 44. `upc_all_gather_all` **for 16 threads over GASNet runtime system**

7.4.5 `upc_all_exchange`

The GMTU implementation for exchange collective communication operation is similar to the gather all collective, where all threads send `nbytes` to their destinations in `THREADS-1` steps and the only difference is that the data items are distinct for each thread, in this case. The results of the exchange collective's comparison are similar to that of gather all, as all threads push or pull `nbytes` of data from all other threads in exchange in a process similar to gather all.

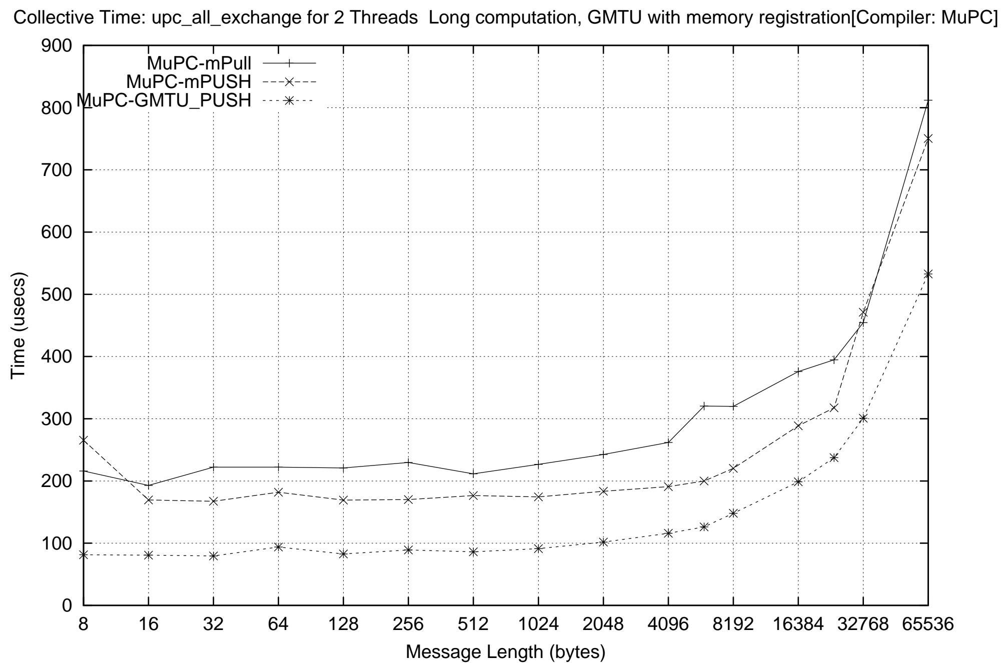


Figure 45. `upc_all_exchange` **for 2 threads over MuPC**

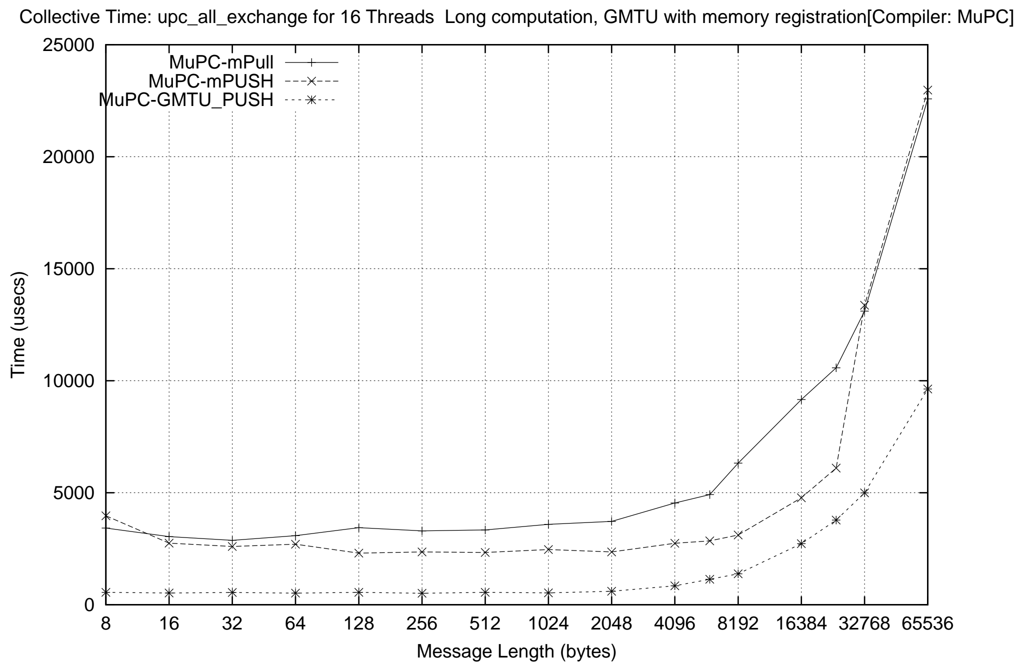


Figure 46. `upc_all_exchange` **for 16 threads over MuPC**

The results for exchange over Berkeley's runtime system are also similar that of gather all in this environment. The results show that for the two 'all-to-all' collective operations, exchange and gather all, the communication patterns are the same. This is true in both the reference implementation and the GMTU memory registration based implementation.

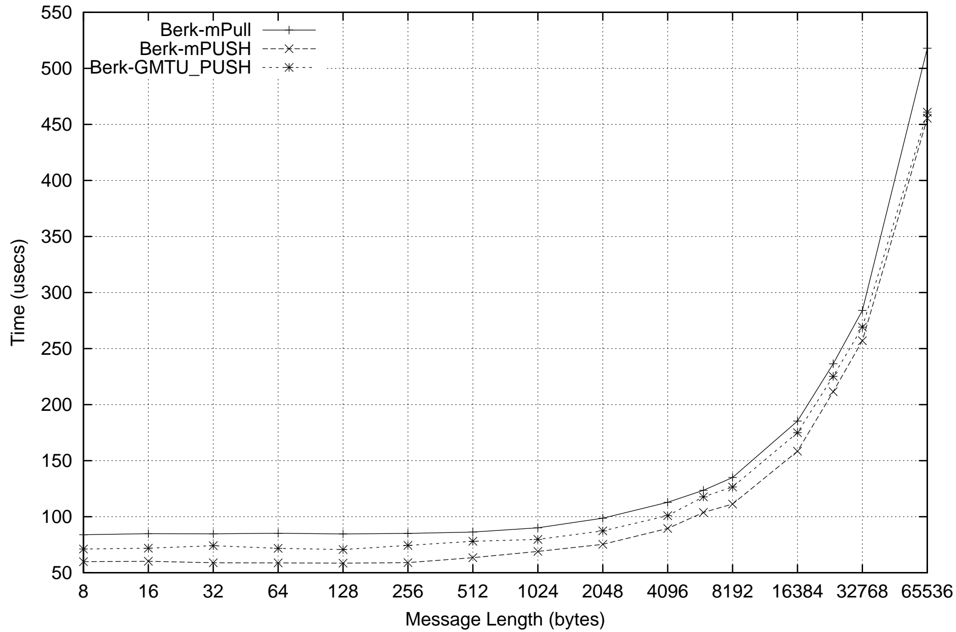


Figure 47. `upc_all_exchange` **for 2 threads over GASNet runtime system**

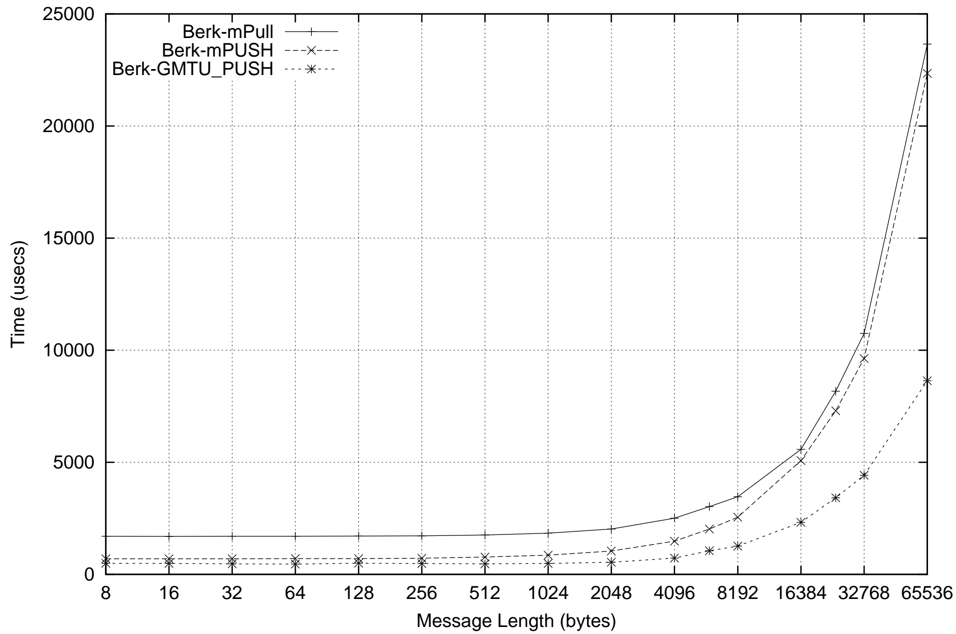


Figure 48. `upc_all_exchange` **for 16 threads over GASNet runtime system**

7.4.6 `upc_all_permute`

The reference implementation for permute in the pull based implementation, involves a loop to search for the source `THREAD` that a destination thread needs to pull the data from. This cost grows with the thread size and is visible when comparing the performance of the pull based implementation for 2 and 16 threads. The push based implementation consists of a source pushing its data to one destination, and over two threads this is the same as an 'all-to-all' communication.

Over 16 threads, the remote put based GMTU implementation performs much faster than the reference implementation, in the MuPC runtime system, because threads perform only one operation and wait at the next synchronization point. The memory registration facility allows this zero-copy performance which results in the observed improvement.

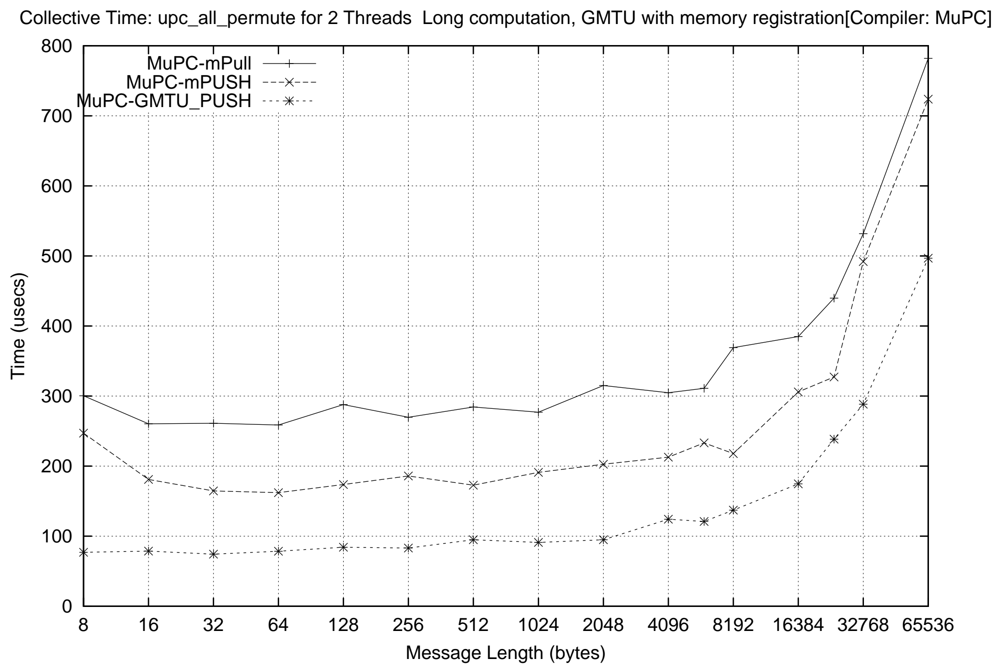


Figure 49. `upc_all_permute` **for 2 threads over MuPC**

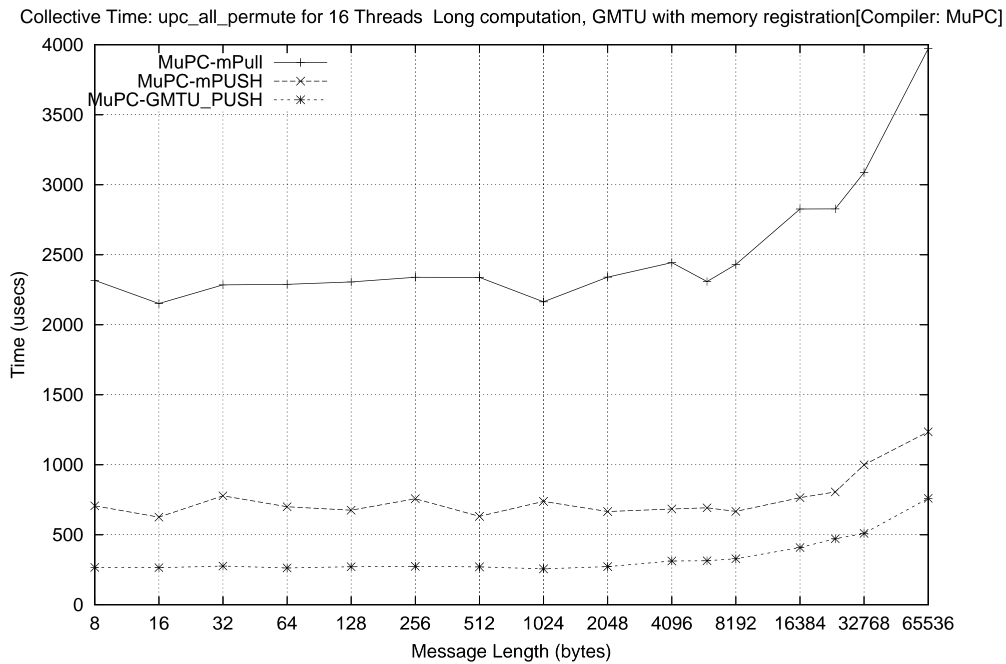


Figure 50. `upc_all_permute` **for 16 threads over MuPC**

In the Berkeley's UPC runtime system based comparisons, we find that the pull based reference implementation continues to be affected by the 'search loop' (for locating the source in the perm array) and the extra overhead due to requesting a remote put from the source. On the other hand the push based reference implementation performs a lot better, however despite using GM's messaging layer the performance of this implementation is not comparable to the GMTU remote put based implementation. This could be due to additional copying or communication overhead imposed by the GASNet layer.

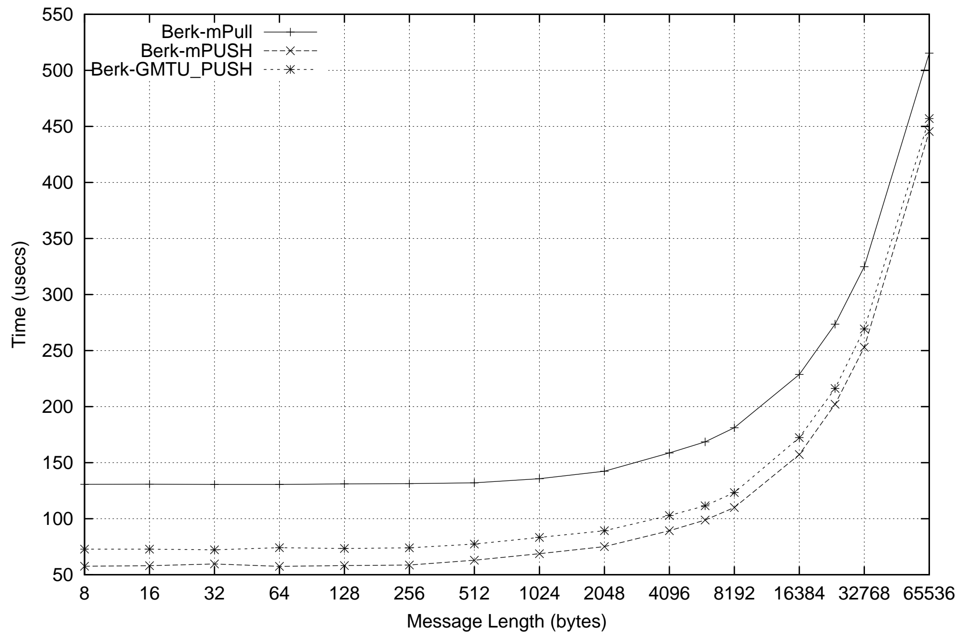


Figure 51. `upc_all_permute` **for 2 threads over GASNet runtime system**

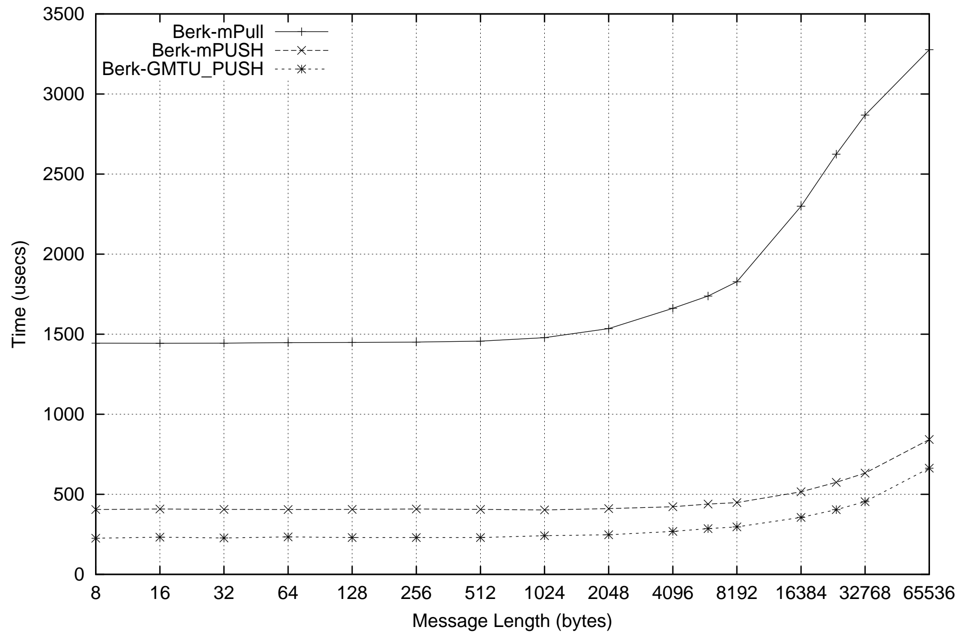


Figure 52. `upc_all_permute` **for 16 threads over GASNet runtime system**

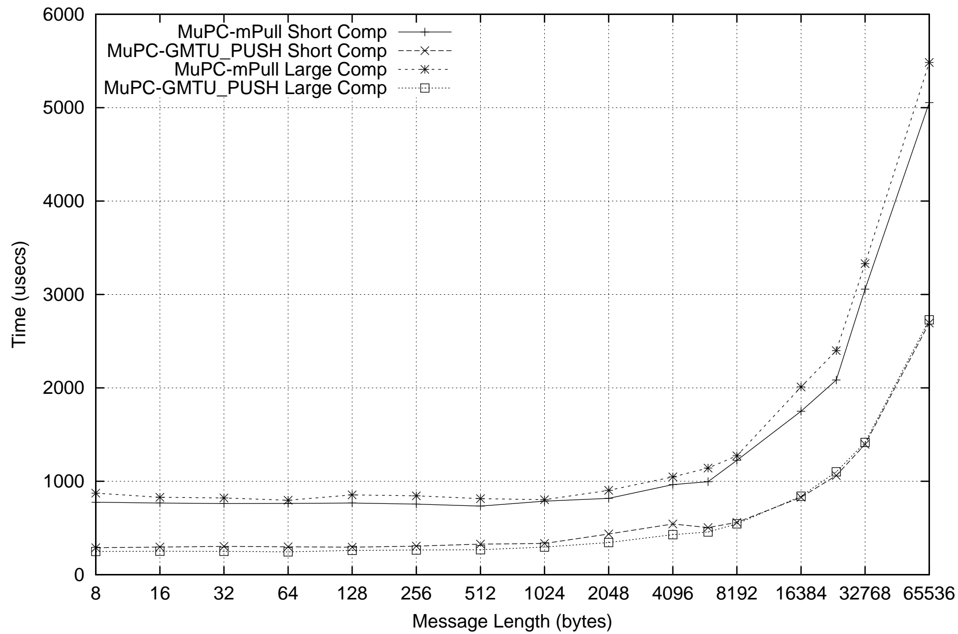
7.5 Effect of Computational Time on Collective Performance

The collective communications were measured using the testbed, with some computation between successive collective calls. The amount of computation and the computational load per thread needed to be studied in more detail to understand their impact on collective operations.

We choose two extreme communication patterns, the broadcast collective representing the 'one-to-all' communication patterns and the exchange collective representing the 'all-to-all' communication pattern. These two collective calls are then measured in both MuPC and GASNet runtime systems for 'short' computation and 'unequal' computation scenarios for 16 threads. These results are then compared with the results from our standard tests, which involve large computation that is uniform over all threads.

7.5.1 Short computation

In the short computation time scenaro, calls to successive collective communication routines had very small computation in between, and this was uniform over all threads. The results indicate that changing the computation time has little effect on the collective communication times, when the computation is uniform. This is true for both, one-to-all and all-to-all collectives.



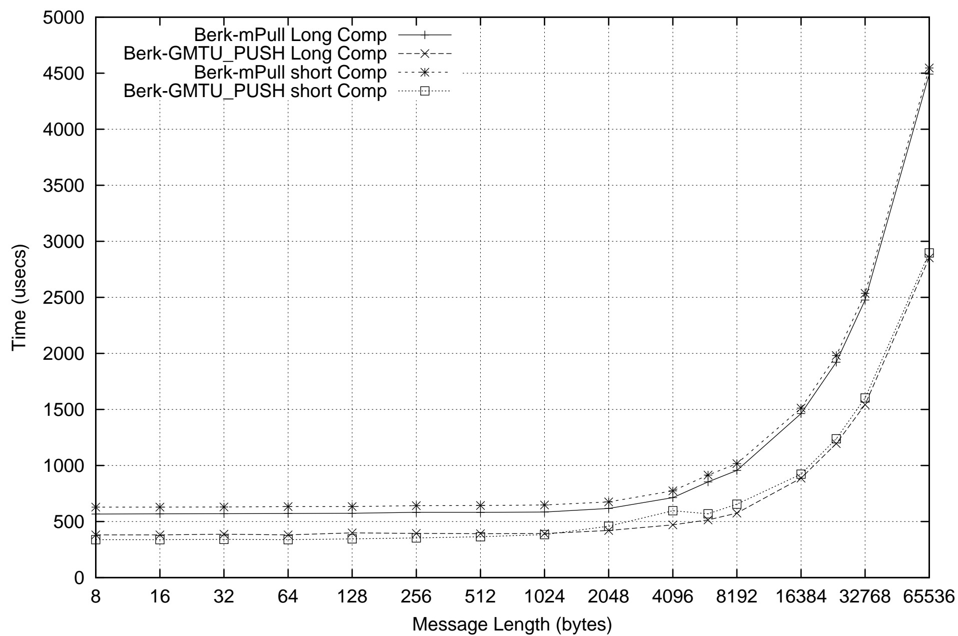
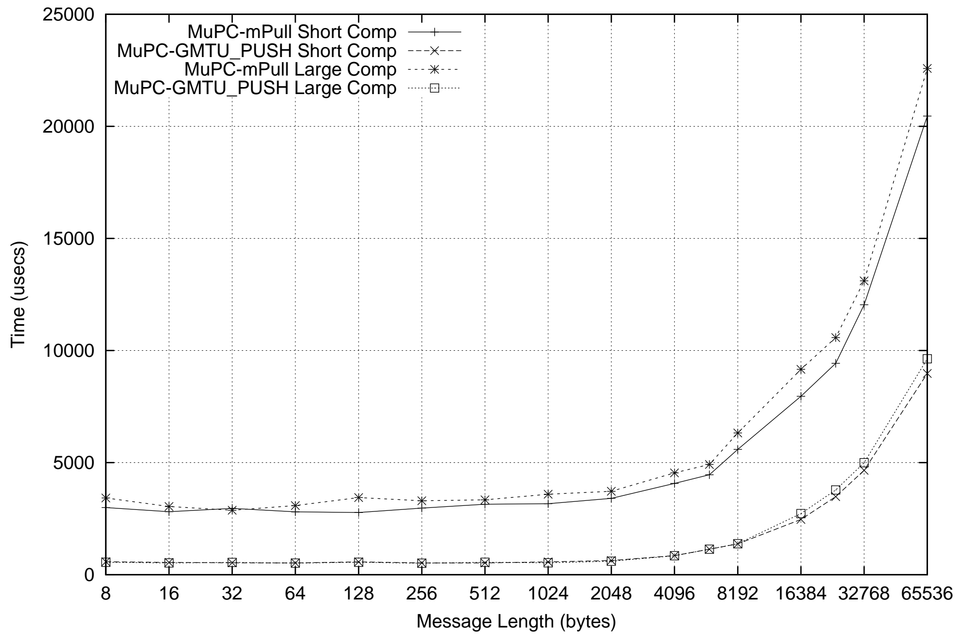


Figure 53. Collective time for short and large computation times,`upc_all_broadcast` over MuPC and Berkeley's UPC



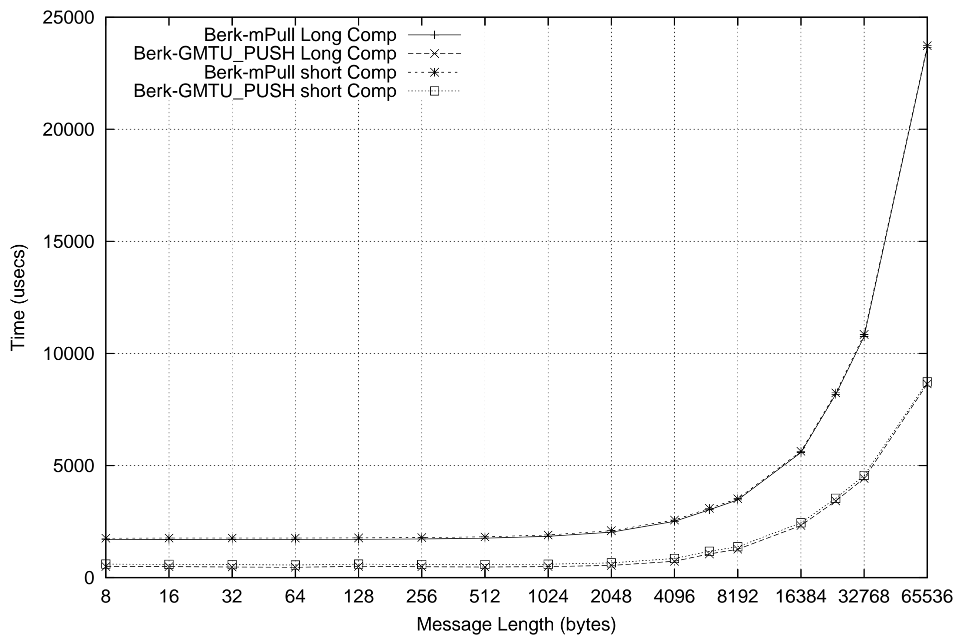
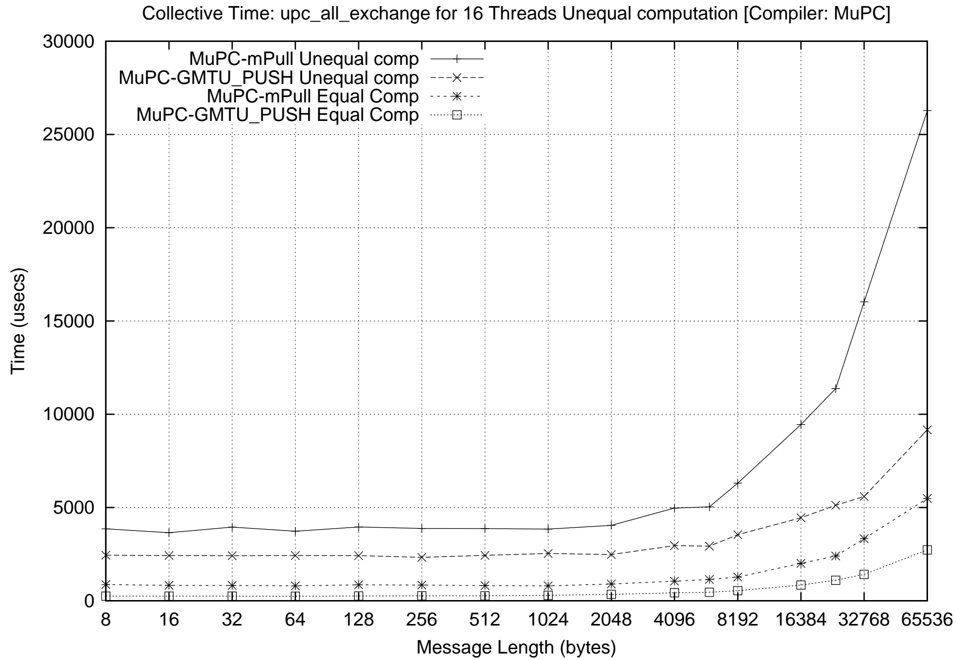


Figure 54. Collective time for short and large computation times,`upc_all_exchange` **over MuPC and Berkeley's UPC**

7.5.2 Unequal computation

In the unequal computation scenario, one thread performed ten times as much computation than the rest of the threads. Also, we ensure that this is not the source thread, as it would invariably delay all the other threads within the collective. The results show that the non-uniform or unequal computation has an effect on the collective communication time. This is expected, as the threads spend longer time waiting at a synchronization step before the collective, for the 'slow' thread. The additional cost due to unequal computation is therefore in the synchronization, and is true for both, one-to-all and all-to-all collectives.



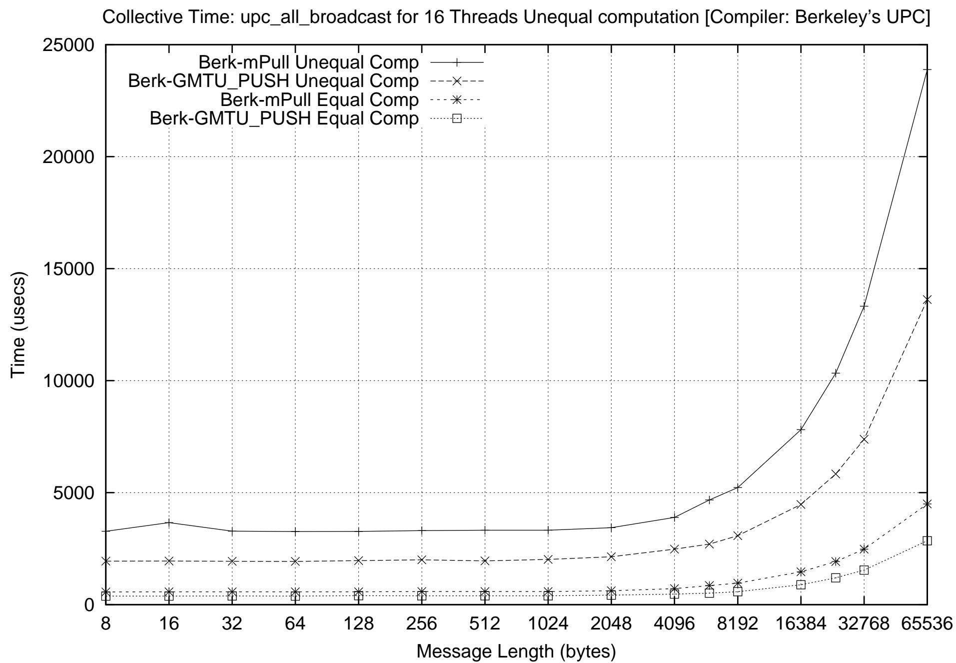
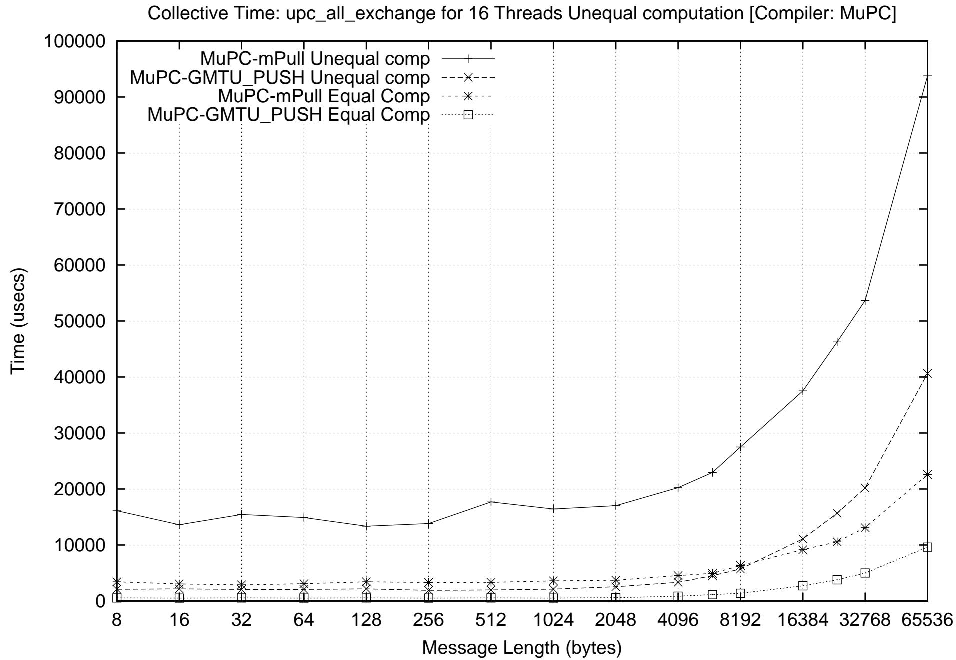


Figure 55. Collective time for unequal and equal computation times,`upc_all_broadcast` **over MuPC and Berkeley's UPC**



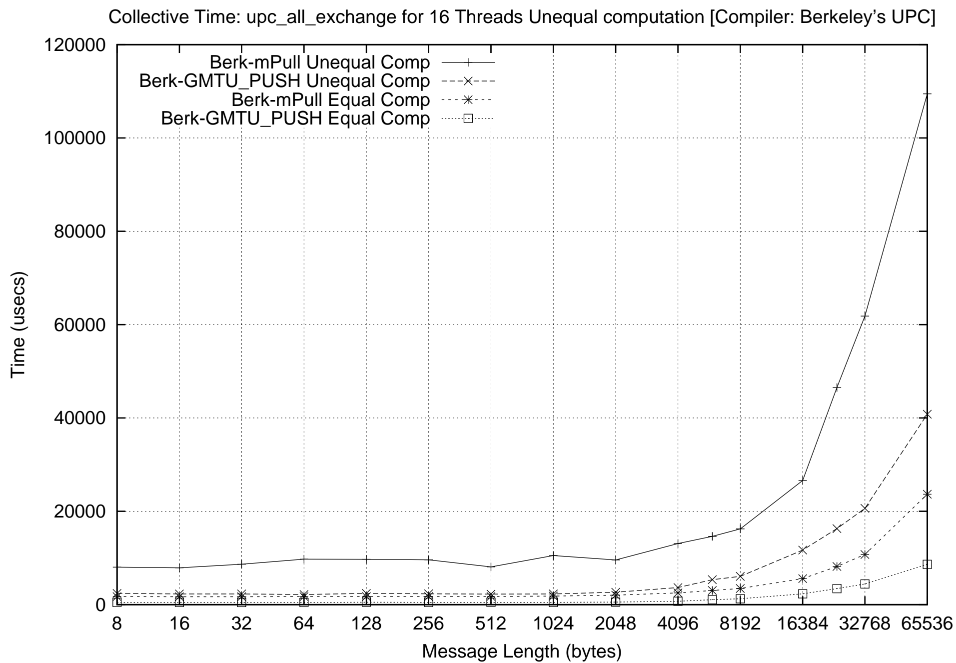


Figure 56. Collective time for unequal and equal computation times,`upc_all_exchange` **over MuPC and Berkeley's UPC**

8 Conclusion

The GM based collective communication library for UPC, provides users with a set of high performance functions for performing relocalization operations in the partitioned shared programming model. As mentioned earlier, the library has been developed to allow the user to select algorithms based on available DMA'able memory for messaging over Myrinet interconnect.

The three main memory management schemes employed by our library are the memory registration, dynamic buffer allocation and static buffer allocation schemes. The memory registration scheme, using `gm_register_memory()` and a registration table of fixed length, uses the rdma puts allowed by GM-1 by resolving the remote memory location of source and destination buffers through UPC's memory model. The collective functions implemented under this scheme perform two to three times better than the reference implementation under the MuPC runtime system, compiled under MPICH-GM. The GMTU implementation of the collectives also perform better than the reference implementation in the Berkeley UPC environment for most collectives and are comparable in performance for the rest.

The collectives implemented in the dynamic and static buffer allocation schemes also perform better than the reference implementation under both MuPC and Berkeley UPC. The static buffer implementation's performance is subject to the size of the static buffer allocated, as small buffer sizes would result in overhead due to extra packetization by the GMTU application.

We also observe that the reference implementation perform better in the Berkeley UPC environment than in MuPC. However, in the naive push/pull case the MuPC runtime system with a cache performs better than the same implementation over Berkeley's GASNet runtime system for UPC.

The results also show the impact of computation time over collective time, which indicate that unequal computation in between successive collective operation results in more time spent during synchronization within the collective calls. This also highlights the synchronization overhead during a collective call, where the actual communication cost could be low but the overall collective time is higher due to the time spent in the synchronization step. This validates the notion of having different synchronization modes in UPC, so that application programmers can choose the synchronization mode for a collective call. For example, when making successive collective calls over unrelated data items, programmers can avoid extra synchronization costs by synchronizing only during the first and the last calls.

The synthetic benchmarking application testbed and its associated test scripts, provide UPC users with a means to compare and measure collective communication libraries. The testbed can be used to measure the performance of collective communications over a wide range of environmental parameters, such as message length, thread size, synchronization, computation

skew etc. We were able to use the testbed to compare and measure different set of collective libraries and were able to observe many interesting results from the different implmentations under different compilers/runtime systems, as stated in the results section.

Our future work for the GMTU collective library would include optimizing the GMTU implemented algorithms for the `UPC_MY_*SYNC` and `UPC_NO_*SYNC` modes. This would also include optimizing the dynamic and static buffer algorithms using remote puts for GM-1, and developing pull based collectives using rdma gets in GM-2. The future work also includes implementing the remaining set of collectives (prefix reduce, sort etc), using the different memory allocation schemes and developing a more scalable alternative for the scatter and gather collectives.

List of Figures

1 The partitioned shared memory model where threads have affinity to regions have the shared address space. 6

2 Creation and allocation of a shared array 'a' with block size `nbytes = 2`, and size `nblocks * nbytes = 10`. Simply put, among 4 threads, the shared array of 10 elements is distributed in blocks of two elements per thread and wrapped around. 7

3 Ports and connections in GM. The messaging is connectionless in GM, meaning no route discovery needs to be done by a sending host - it simply says 'send this message to this node id through my this port' and the message is sent.[**?**] . 10

4 Tokens are used to keep track of sent/received messages by GM hosts. [*Source* : *Myricom* [**?**]] 11

5 When sending a message in GM the message priority and size must be the same as that expected by the receiver.[*source* : *Myricom* [**?**]] 13

6 When receiving messages in GM, we must first provide a receive buffer of matching size and priority as the sender did while sending this message. Then we must poll for the total number of expected. [*Source* : *Myricom* [**?**]] 14

7 Referencing shared address space that a thread does not have affinity to, can be costly. This picture shows the `nbytes`-sized block of a char array `b` (shaded) that thread 0 has affinity to. 15

8 Post-broadcast we see that a copy of `nbytes` bytes of char array `b` have been sent to each thread and are stored in array `a`. 16

9 `upc_all_broadcast` using 'push' and 'pull' based protocols. (a) Push based collectives rely on one thread copying the data to the rest, while (b) pull based collectives are more efficient and parallelized as destination threads are responsible to copying their own data from the source. 17

10 upc_all_broadcast push implementation 19

11 upc_all_scatter push implementation 20

12 upc_all_gather push implementation 21

13 upc_all_gather_all push implementation 21

14 upc_all_exchange push implementation 22

15 upc_all_permute push implementation 23

16 (a) Collectives as measured between computations in traditional benchmarks, computation and collective communication times are not proportional. (b) Collectives as measured by our benchmark, collective and computation times are kept proportional 25

17 `upc_all_broadcast` using memory registration, and rdma put for a push based implementation using GM 32

18 `upc_all_gather` using memory registration and remote puts 37

19 (a) using logical ring for dynamic and static DMA allocation based `upc_all_gather_all` algorithms (b) Using remote puts, rdma write, for memory registration based `upc_all_gather_all` algorithm 40

20 GM based barrier using gather and broadcast collectives for a short message length 43

21 GMTU test matrix 53

22 Naive push/pull vs reference implementation, MuPC and GASNet, and GMTU memory registration based push 56

23 `upc_all_broadcast` for 16 threads with GMTU memory registration, dynamic and static allocation 57

24 `upc_all_scatter` for 16 threads with GMTU memory registration, dynamic and static allocation 58

25 `upc_all_gather` for 16 threads with GMTU memory registration, dynamic and static allocation 59

26 `upc_all_gather_all` for 16 threads with GMTU memory registration, dynamic and static allocation 60

27 `upc_all_exchange` for 16 threads with GMTU memory registration, dynamic and static allocation 61

28 `upc_all_permute` for 16 threads with GMTU memory registration, dynamic and static allocation 62

29 `upc_all_broadcast` for 2 threads over MuPC 64

30 `upc_all_broadcast` for 16 threads over MuPC 65

31 `upc_all_broadcast` for 2 threads over GASNet runtime system 66

32 upc_all_broadcast for 16 threads over GASNet runtime system 66
33 upc_all_scatter for 2 threads over MuPC 67
34 upc_all_scatter for 16 threads over MuPC 68
35 upc_all_scatter for 2 threads over GASNet runtime system 69
36 upc_all_scatter for 16 threads over GASNet runtime system 69
37 upc_all_gather for 2 threads over MuPC 70
38 upc_all_gather for 16 threads over MuPC 71
39 upc_all_gather for 2 threads over GASNet runtime system 72
40 upc_all_gather for 16 threads over GASNet runtime system 72
41 upc_all_gather_all for 2 threads over MuPC 73
42 upc_all_gather_all for 16 threads over MuPC 74
43 upc_all_gather_all for 2 threads over GASNet runtime system 75
44 upc_all_gather_all for 16 threads over GASNet runtime system 75
45 upc_all_exchange for 2 threads over MuPC 76
46 upc_all_exchange for 16 threads over MuPC 77
47 upc_all_exchange for 2 threads over GASNet runtime system 78
48 upc_all_exchange for 16 threads over GASNet runtime system 78
49 upc_all_permute for 2 threads over MuPC 79
50 upc_all_permute for 16 threads over MuPC 80
51 upc_all_permute for 2 threads over GASNet runtime system 81
52 upc_all_permute for 16 threads over GASNet runtime system 81
53 Collective time for short and large computation times,upc_all_broadcast over MuPC and Berkeley's UPC 83
54 Collective time for short and large computation times,upc_all_exchange over MuPC and Berkeley's UPC 84
55 Collective time for unequal and equal computation times,upc_all_broadcast over MuPC and Berkeley's UPC 86

56 Collective time for unequal and equal computation times,`upc_all_exchange` over MuPC and Berkeley's UPC 87