Implementing UPC's MYSYNC Synchronization Mode Using Pairwise Synchronization of Threads

Prasad Dhamne and Steve Seidel

Michigan Technological University

Computer Science Technical Report

CS-TR-05-07

2005

Abstract

UPC (Unified Parallel C) is an extension of ANSI C that provides a partitioned shared memory model for parallel programming. Synchronization between threads (processes) in UPC is done through the use of locks or barriers. We have investigated a new synchronization method which is better suited in situations where threads do not need to synchronize with all of the other threads in the system.

We implemented pairwise synchronization that can be used to synchronize pairs of threads while not disturbing the remaining threads. This project also explored how to synchronize pairs of UPC threads over a Myrinet interconnect. Myrinet is a low-latency, high bandwidth local area network with a low level communication layer called GM. We implemented UPC's `MYSYNC` synchronization mode in collective operations which make use of pairwise synchronization. We compared the performance of `MYSYNC` synchronization mode to the `ALLSYNC` synchronization mode, that is often implemented by using a barrier. For performance analysis we used a collectives testbed previously developed at MTU.

Results obtained from the testbed indicate that for collectives, such as `Broadcast`, `Scatter` and `Gather`, `MYSYNC` synchronization considerably reduces the collective time. Overall testbed execution time for 100 trials with `MYSYNC` mode was improved by 10-15% compared to the `ALLSYNC` synchronization. For the `Permute` collective operation the performance improvement is maximum in pull based implementation and it is close to 20%. For `Gatherall` and `Exchange` the performance improvement is minimal. However, `ALLSYNC` mode performs better over `MYSYNC` synchronization in push based implementation of `Permute` collective operation. In general, the performance of `MYSYNC` reduces with increase in number of threads.

Contents

List of Figures

Chapter 1

Introduction

Unified Parallel C (UPC) is a parallel programming language based on a partitioned `shared` memory model that is being developed by a consortium of academia, industry and government [2,4,7]. It is an extension of ANSI C, aimed for high performance computing on various platforms. In UPC, each participating thread is an independent process and has a `shared` address space along with a local address space. 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 are costlier (and similarly accesses to regions which have affinity to the thread are less costly) [4]. UPC allows the programmer to change the affinity of `shared` data to better exploit its locality through a set of collective communication operations.

UPC provides keywords and methods for parallel programming over the partitioned `shared` model. It includes the `shared` type specifier, synchronization mechanisms such as locks, barriers and memory consistency control (strict or relaxed). UPC is intended to be a simple and easy to write programming language where remote reads and writes are done implicitly. This enables the user to concentrate more on the parallelization task, rather than worrying about the underlying architecture and communication operations 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` type specifier. For instance, creation and allocation of a `shared` array D with 9 elements, block size 2 and 4 `THREADS` is declared by simply putting, amongst 4 threads, the `shared` array of 9 elements in blocks of 2 elements per thread and wrapped around. Figure 1.1 shows different ways of using the *shared* qualifier and resulting distribution of data in the `shared` memory space [2].

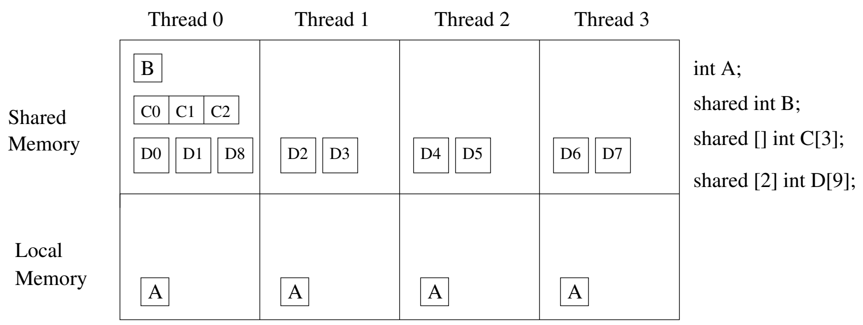


Figure 1.1: UPC memory model and Data distribution

1.1 MuPC

Originally UPC was developed for vector machines, such as the Cray T3E, and with the popularity of cluster based high performance computing new compilers, runtime systems and libraries were required to extend the language to these architectural models. Michigan Tech's MuPC is one such runtime system that uses MPI to bring UPC to platforms like Beowulf clusters that support a partitioned `shared` memory environment [4]. MuPC relies on the MPI communication library to implement reads and writes in `shared` memory. Depending upon the MPI compiler, we can choose the underlying interconnect. For example, the Beowulf cluster lionel in MTU's Center for Experimental Computing (CEC), has both Ethernet and Myrinet interconnects. MPI programs can be compiled under either LAM-MPI or MPICH-GM to use the Ethernet or Myrinet interconnects respectively. A user can write supporting libraries for UPC just as in C, for mathematical calculations, string manipulation and to add additional features such as pairwise synchronization of threads or `MYSYNC` synchronization mode in collective operations.

1.2 GM

The Myrinet network is a low-latency, high bandwidth local area network developed by Myricom [6]. Compared to conventional networks such as Gigabit Ethernet, Myrinet provides features for high-performance computing using full-duplex links, flow control, cut-through routing and OS-bypass. It is a robust, highly scalable interconnect and comes with a communication API, called GM. GM provides a notion of logical software ports which are different 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 applications. The maximum transmission unit (MTU), which defines the size of the largest packet that can be physically sent, is 4K or 4096 bytes for GM. During transmission all packets of size larger than the MTU are broken down to around the 4K size before being sent. The GM API provides various functions to send and receive messages using the GM communication layer. These functions can be used to speedup the performance of user applications. We are using some of the send/receive functions to implement data transfer among the collectives.

1.3 Motivation

The motivation behind this project is the highly efficient GM based collective communication library developed by Mishra [4]. It makes use of GM to do low level message passing among the UPC threads. This library achieves a 40-50% performance improvement over the reference implementation of UPC collectives. At present in both these libraries the `MYSYNC` synchronization mode is implemented with a barrier which acts as bottleneck to performance. A previous project of Michigan tech to implement a `MYSYNC` synchronization using `UPC` had unpredictable performance improvement [5].

The other motivation behind this project is to prove that `MYSYNC` synchronization offer performance benefits compared to full barrier synchronization. Some researchers suggest that the `MYSYNC` synchronization mode is unnecessary. We found some applications where `MYSYNC` mode could give better performance than `ALLSYNC` mode. We are aiming to compare our implementation of the `MYSYNC` mode with the `ALLSYNC` mode and want to demonstrate that `MYSYNC` mode provides performance benefits.

Chapter 2

Collective Communication in UPC

2.1 Need for Collective Communication Operations

As UPC is a shared memory programming language, each thread has local address space and shared address space. Each thread has affinity to its own shared space which relates that partition of shared space being closer to that thread as compared to other threads. 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 2.1). The execution time of application increases if each thread frequently access the shared space to which it has no affinity. Hence, data relocalization is required which reduces the access to remote shared memory and in turn reduces the application execution time. Collective operations provide a way to achieve this relocalization of data, by changing the affinity of the data item. An example of how the broadcast function performs relocalization is illustrated below [4]:

- 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

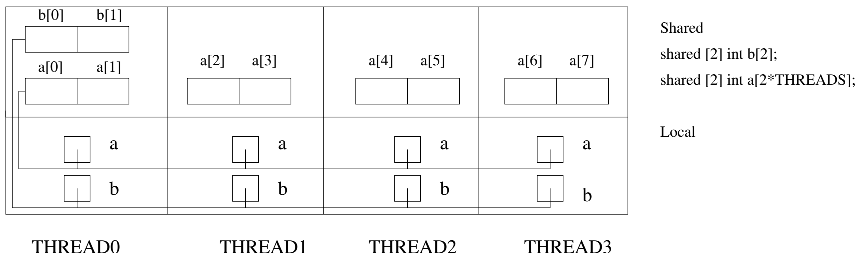


Figure 2.1: Pre-broadcast data layout

- 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`

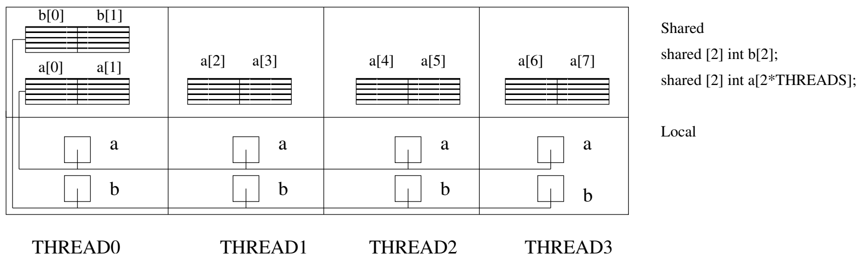


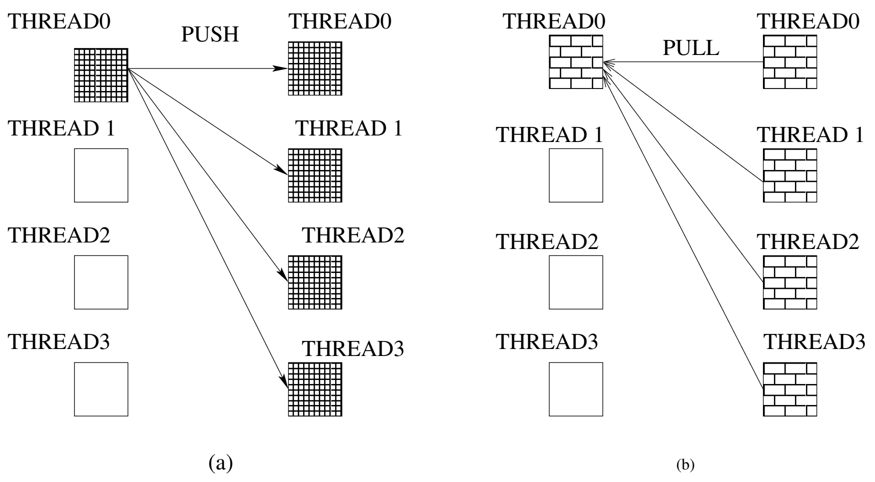
Figure 2.2: Post broadcast data layout

2.2 Push and Pull Implementations

We know collective communication operations play an important role in the relocalization of data. Before we go on to the types of collective operations we must first understand the two basic ways in which data is distributed. Consider a producer consumer arrangement [4], where a producer produces goods that are consumed by many consumers (1 producer and more than one consumer). 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 and nodes and chunks 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 nodes simply read a memory area from the source? Therefore, collective communication operations where source thread(s) send the data to the destination thread(s) are the `push` based implementation; while operations where the destination thread(s) read the data from the source thread(s) are termed `pull` based implementations. Figure 2.3 shows how `push` and `pull` broadcast operations differ. In Figure 2.3(a), thread 0 is writing the data to the destination threads by copying it into their memory locations; whereas in (b) each destination thread is responsible for reading the data from thread 0's memory. 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 [3]. 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 destination 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 [4].

- Pseudocode for push reference implementation of `upc_all_broadcast`

```
begin upc_all_broadcast(shared void *destination,
                        shared const void *source,
```



(a) Push based collectives rely on source thread copying data to each destination
(b)In Pull based collectives each destination thread copies data to source thread

Figure 2.3: Push-Pull implementation of collectives

```
                        size_t nbytes,
                        upc_flag_t sync_mode)
source:= upc_threadof 'source' array
if(MYTHREAD = source) then
 for i:=0 to THREADS
   upc_memcpy 'nbytes' from 'source' into 'destination+i'
 end for
end if
end upc_all_broadcast
```

- Pseudocode for pull reference implementation of `upc_all_broadcast`

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

2.3 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* [3]. The details of these collective functions, their parameters, and their operations on shared arrays are described below.

- **upc_all_broadcast**

  `void upc_all_broadcast(shared void *destination, shared const void *source, size_t nbytes, upc_flag_t sync_mode)`

  Description:

  The source pointer is interpreted as:

  `shared [] char[nbytes]`

  The destination pointer is interpreted as:

  `shared [nbytes] char[nbytes*THREADS]`

  The function copies nbytes of the source array into each nbyte-block of the destination array.

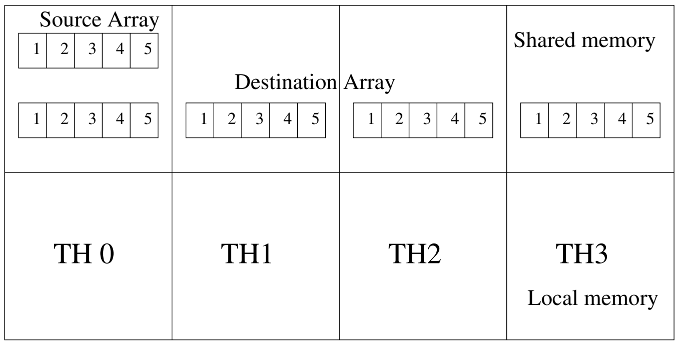


Figure 2.4: upc_all_broadcast

- **upc_all_scatter**

  `void upc_all_scatter(shared void *destination,shared const void *source, size_t nbytes, upc_flag_t sync_mode)`

  Description:

  The source pointer is interpreted as:

  `shared [] char[nbytes*THREADS]`

  The destination array is interpreted as declaring:

  `shared [nbytes] char[nbytes*THREADS]`

  The i^{th} thread copies the i^{th} nbyte-block of the source array into the i^{th} nbyte-block of the destination array which has affinity to the i^{th} thread.

- **upc_all_gather**

  `void upc_all_gather (shared void *destination,shared const void *source, size_t nbytes, upc_flag_t sync_mode)`

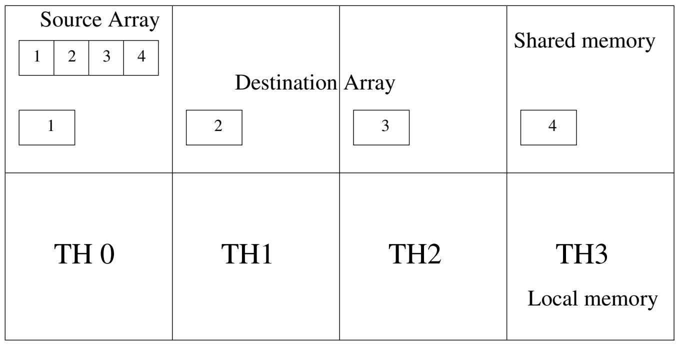


Figure 2.5: upc_all_scatter

Description:

The source pointer is assumed to be an array, declared as:

`shared [nbytes] char[nbytes * THREADS]`

The destination pointer is assumed to be declared as:

`shared [] char[nbytes* THREADS]`

The i^{th} thread copies the i^{th} nbyte-block of the source array, with affinity the i^{th} thread, into the i^{th} nbyte-block of the destination array.

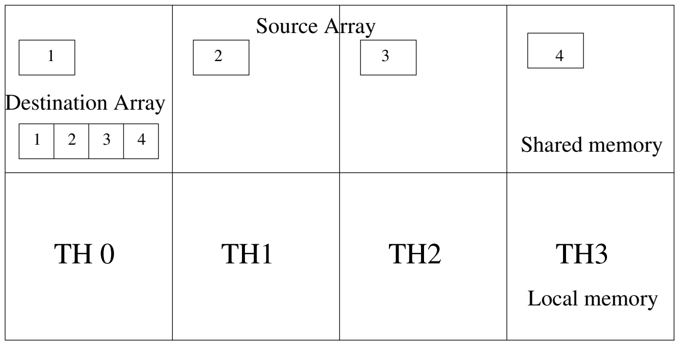


Figure 2.6: upc_all_gather

- **upc_all_gather_all**

  `void upc_all_gather_all (shared void *destination,shared const void *source, size_t nbytes, upc_flag_t sync_mode)`

  Description:

  The source pointer is assumed to be an array, declared as:

  `shared [nbytes] char[nbytes*THREADS]`

  The destination pointer as:

  `shared [ nbytes*THREADS] char[nbytes*THREADS *THREADS]`

  The i^{th} thread copies the i^{th} nbyte-block of the source array into the i^{th} nbyte-block of the destination array.

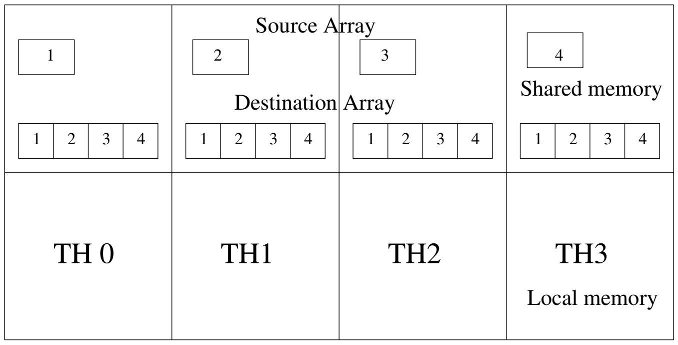


Figure 2.7: upc_all_gather_all

- **upc_all_exchange**

  `void upc_all_exchange (shared void * destination, shared const void *source, size_t nbytes, upc_flag_t sync_mode)`

  Description:

  The source and destination pointers are assumed to be arrays, declared as:

  `shared [nbytes*THREADS] char[nbytes*THREADS*THREADS]`

  The i^{th} thread, copies the j^{th} nbyte-block of the source array into the i^{th} nbyte-block of the destination array which has affinity to the j^{th} thread.

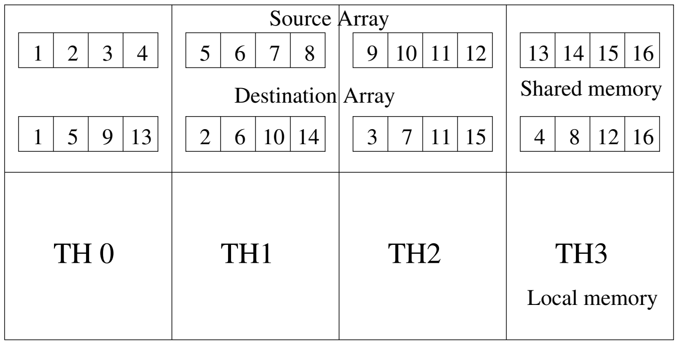


Figure 2.8: upc_all_exchange

- **upc_all_permute**

  `void upc_all_permute (shared void *destination, shared const void *source, shared const int *perm, size_t nbytes, upc_flag_t sync_mode)`

  Description:

  The source and destination pointers are assumed to be char arrays that are declared as:

`shared [nbytes] char[nbytes*THREADS]`

The i^{th} thread, copies the i^{th} nbyte-block of the source array into the nbyte-block of the destination array which has affinity to the thread corresponding to the i^{th} element of the `perm` array.

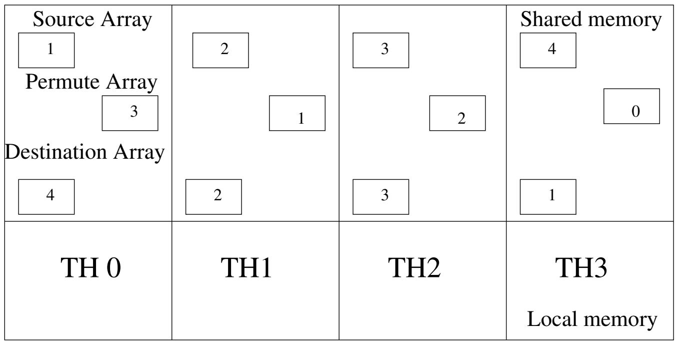


Figure 2.9: upc_all_permute

Chapter 3

Message passing with GM

GM is a message-based communication system that uses the Myrinet interconnect to send and receive messages over a connected network. GM provides low CPU overhead, portability, low latency and high bandwidth. GM also provides reliable ordered delivery between hosts in the presence of network faults. GM detects and retransmits lost and corrupted packets. GM also reroutes packets around network faults when alternate routes exists. In the presence of catastrophic network errors, such as crashed hosts or disconnected links, the undeliverable packets are returned to the client with an error indication. GM provides two levels of message priority to allow efficient deadlock free bounded-memory forwarding. GM allows clients to send messages up to $2^{31} - 1$ bytes long, under operating systems that support sufficient amounts of DMAable memory to be allocated. At the application level, the communication model in GM is connectionless. This means that, unlike other systems that use protocols to set up a communication link prior to message passing, no handshaking is required by GM applications.

3.1 Programming model

GM applications use a message passing communication model similar to MPI. Before calling any other GM function, `gm_init()` should be called. `gm_finalize()` should be called after all other GM calls and before program exits. Each call to `gm_init()` should be balanced by a call to `gm_finalize()` before the program exits. The calls to `gm_finalize()` are required for proper shutdown of GM to allow ports to be reused. A GM programming model requires that every send operation on a source node must have matching receive operation on destination node. These send and receive calls must also match in buffer size and message priority types at both ends. This helps to distinguish between incoming messages. For instance, while sending $nbytes$ bytes 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.

3.1.1 Token flow in GM

Both send and receives in GM are regulated by implicit tokens. These tokens represent space allocated to the client in various internal GM queues, as depicted in the Figure 3.1. At the initialization of GM the client implicitly possesses `gm_num_send_tokens()` send tokens, and `gm_num_receive_tokens()` receive tokens.

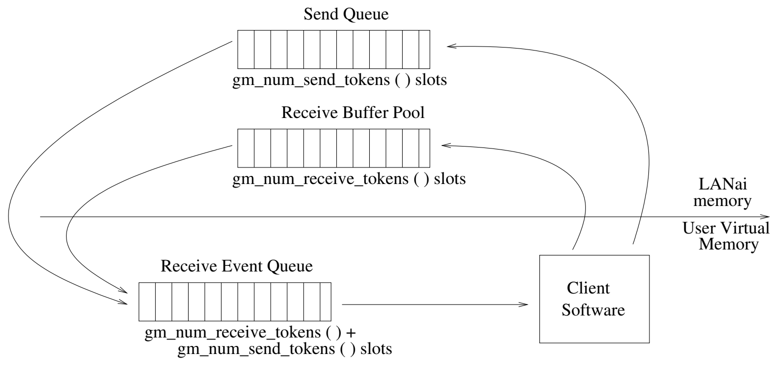

Send Queue
gm_num_send_tokens ( ) slots
Receive Buffer Pool
gm_num_receive_tokens ( ) slots
LANai memory
User Virtual Memory
Receive Event Queue
gm_num_receive_tokens ( ) + gm_num_send_tokens ( ) slots
Client Software

Figure 3.1: Tokenflow in send/receive operations in GM

In order to issue either a send or receive operation the client has to make sure that it has a token. After issuing the send operation the `send` client implicitly relinquishes the token, which is returned back to it when the send completes. Similarly, a call to `gm_provide_receive_buffer` at the `receive` client will release a receive token and once a matching receive (based on size and priority) has been received token will be returned to client. Calling a GM API function without the required tokens has undefined results, but GM usually reports such errors.

3.1.2 DMA allocation schemes in GM

All messages sent and received by GM must reside in DMAable memory. GM API provides functions such as `gm_dma_malloc()` and `gm_dma_calloc()` to allocate new DMAable memory. 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 and adds a page table entry to a DMAable page table that LANai accesses enabling GM to read/write onto that region [4]. 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.

3.2 Sending messages in GM

As mentioned earlier, to send messages the client application must keep track of the number of send tokens it has. Before making calls to any GM functions that require tokens, the client application should call `gm_num_send_tokens()` to make sure that it possess the required token. `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. By calling a GM send function, the client relinquishes that send token. The client passes a callback and context pointer to the send function as shown in Figure 3.2. When the send completes, GM calls callback, passing a pointer to the GM port, the client-supplied context pointer, and the status code indicating if the send completed successfully or with an error. When GM calls the client's callback function, the send token is implicitly passed back to the client.

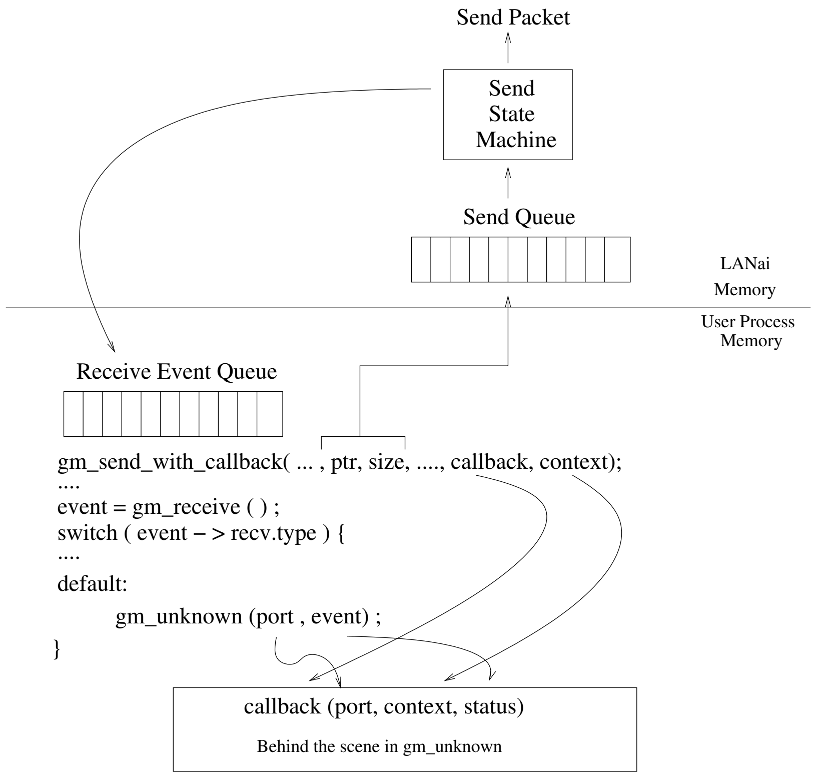

Send Packet
Send State Machine
Send Queue
LANai Memory
User Process Memory
Receive Event Queue
gm_send_with_callback(... , ptr, size, ..., callback, context);
....
event = gm_receive ( ) ;
switch (event – > recv.type) {
....
default:
gm_unknown (port , event) ;
}
callback (port, context, status)
Behind the scene in gm_unknown

Figure 3.2: Sending message in GM

Most GM programs that rely on GM's fault tolerance to handle transient network faults, should consider a send that completed with a status other than `GM_SUCCESS` to be a fatal error. It is also important to note that the sender does not need 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.

3.3 Receiving messages in GM

GM receives are also regulated by the notion of implicit tokens. After a port is opened, the client implicitly possesses `gm_num_receive_tokens()` receive tokens, allowing it to provide GM with up to this many receive buffers using `gm_provide_receive_buffer()`. Before a receiving client can receive a message intended for it, it must provide a receive buffer of the expected size and priority. With each call to `gm_provide_receive_buffer()` the receiving client gives up a receive token as shown in Figure 3.3. This token is returned after a receive event has been handled properly. 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 receive event types that may be generated depending on the size, priority and receive port of the receive event type.

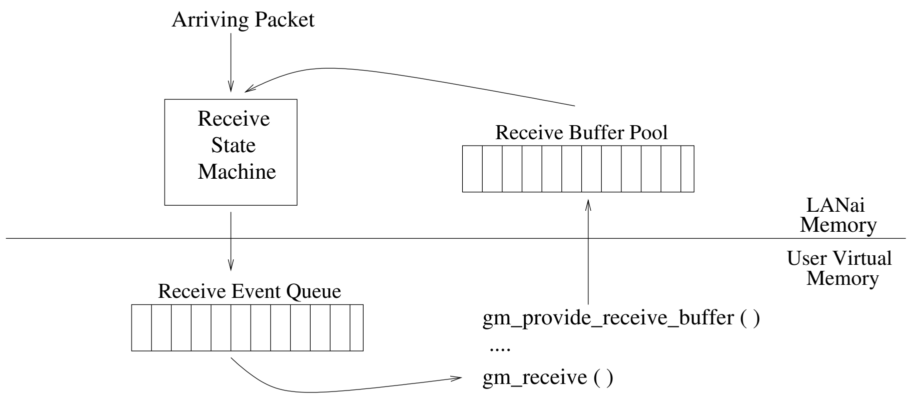


Figure 3.3: Receiving message in GM

The client application handles all the expected receive event types as shown in the following `algorithm`[6] 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.

```
gm_provide_receive_buffer(g_ptr->my_sync_port,g_ptr->sync_buffer,
                          gm_min_size_for_length(nbytes),
                          GM_HIGH_PRIORITY);
messages=1;
while(messages)
{
  event = gm_receive(g_ptr->my_sync_port);
  switch (gm_ntoh_u8(event->recv.type))
  {
    case GM_RECV_EVENT
      if(g_ptr->node_list[source]==gm_ntohs(event->recv.sender_node_id))
         messages--;
      else
         gm_provide_receive_buffer(g_ptr->my_sync_port,g_ptr->sync_buffer,
                                   gm_min_size_for_length(nbytes),
                                   GM_HIGH_PRIORITY);
    break;

    case GM_NO_RECV_EVENT:
    break;

    case default:
    gm_unknown(g_ptr->my_sync_port,event);/*calls callback*/
  }
}
```

Chapter 4

Synchronization modes in UPC

4.1 Synchronization Modes

All of the computational and relocalization collective operations in UPC have two synchronization modes specified by the single argument `sync_mode`. These modes enforce guarantees on the state of the data before and after the collective call. The following text describes the synchronization modes in detail [3]. If `sync_mode` has the value (`UPC_IN_XSYNC` |`UPC_OUT_YSYNC`), then

If `X` is,
NO : the collective function may begin to read or write data when the first thread has entered the collective function call
MY : the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call
ALL : the collective function may begin to read or write data only after all threads have entered the collective function call
If `Y` is,
NO : the collective function may read and write data until the last thread has returned from the collective function call
MY : the collective function call may return in a thread only after all reads and writes of data with affinity to the thread are complete
ALL : the collective function call may return only after all reads and writes of data are complete.

The `NOSYNC` synchronization mode is intended to be used when the programmer knows that the running threads are not dependent on the data being sent or received [5]. This synchronization mode should also be used if the programmer knows that there is more computation that can be done after the collective operation that does not use the data being sent or bundles many disjoint collective operations together and begins as well as ends the sequence with a barrier. The `ALLSYNC` synchronization mode is the easiest for the programmer because it provides the equivalent of barrier synchronization, and the programmer knows that no thread will interfere with the data being sent or received.

It is the work of this project to show that the `MYSYNC` synchronization mode should be used when the programmer knows that each thread will work on the data it has received and will not interfere with any other thread's data. Our interest is in the implementation and performance of the `MYSYNC` synchronization mode compared to a barrier implementation of the `ALLSYNC` synchronization mode. As the collective specification has developed there was a large amount of debate as to whether all three synchronization modes were needed. There is currently a debate as to whether the `MYSYNC` synchronization mode will be used at all and whether it promises any performance gain. The goal of this project is to show that the `MYSYNC` does have its place in the collective specification and will have better performance than the `ALLSYNC` synchronization mode in some situations.

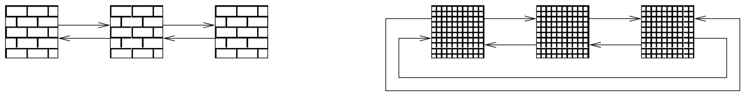
4.2 Pairwise Synchronization

The pairwise synchronization process involves a pair of threads which communicate in order to synchronize with each other. Ideally, threads involved in pairwise synchronization do not affect the performance of other threads. We can use pairwise synchronization to implement `upc_barrier` or to implement synchronization among an arbitrary subset of threads.

4.2.1 Applications

Pairwise synchronization may be more efficient in some cases because it allows the earlier arriving threads to synchronize first, allowing them to exit the synchronization step sooner, so they can continue with other work. These threads would be able to synchronize with whatever threads they need to, provided those other threads are ready, and not have to wait for all of the other threads in the system. Certain threads may arrive at a collective call earlier than others due to several factors, primarily the work environment as other users of the computer may have jobs running taking up CPU cycles. Also, some algorithms may have threads that perform different tasks than the other threads in the system. This type of synchronization is desirable for algorithms where the processors only need to synchronize with their immediate neighbors as some following examples illustrate.

By using pairwise synchronization we can construct routines that are capable of synchronizing any given set of threads. Applications that are set up in a pipeline or ring fashion are able to synchronize with their nearest neighbors without forcing all of the processors to synchronize at a given point [5]. Having thread 0 and thread 2 synchronize is not needed as thread 0 and thread 2 do not have to communicate in a pipeline. Figure 4.1 illustrates these ideas. Each box represents a thread and the arrows represent the synchronization.



(a) Pipeline Pattern (b) Ring Pattern

Figure 4.1: Pipeline and ring based synchronization patterns

Pairwise synchronization is also applicable to grids [5]. In grid algorithms where each thread only synchronizes with its immediate neighbors there is no need for threads to synchronize with threads that are not its immediate neighbors. If we use the UPC barrier then those threads would be synchronized. This also applies to algorithms that use a torus communication patterns.

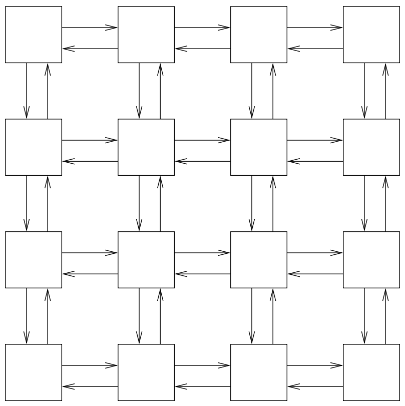


Figure 4.2: Grid based synchronization patterns

Figure 4.2 illustrates this scenario in which each thread issues pairwise synchronization calls which is equal to size of its neighbor set.

Using pairwise synchronization we can also set up a routine to facilitate master/ drone algorithms such as the LU decomposition version of the matrix multiplication algorithm. This allows the master to synchronize only with the drones that need more work, and not with the entire system [5]. Figure 4.3 illustrates a possible synchronization pattern in a master drone algorithms.

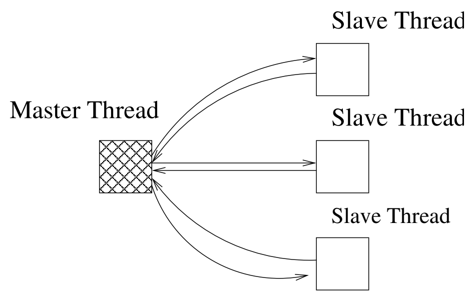


Figure 4.3: Synchronization scheme for master drone algorithms

Chapter 5

Project work

The main goal of this project work is to show that `MYSYNC` synchronization in UPC collectives provides better performance than `ALLSYNC` synchronization in certain scenarios. With this in mind, we implemented push and pull based `UPC` level `MYSYNC` synchronization and compared it with respective reference implementations of all 6 relocalization collectives. We also implemented push based `GM` level synchronization and compared it with Mishra's push based `GM` level implementation of `ALLSYNC` synchronization. Finally, we integrated our implementation of `MYSYNC` synchronization with the reference implementation of `ALLSYNC` and `NOSYNC` synchronization so that user can use our library for any combination of `UPC_IN_*SYNC` and `UPC_OUT_*SYNC` (Figure 6.28) to specify synchronization at the time of entering and leaving the collective operation. The performance evaluation is done by using the testbed developed at MTU. The details of this work are provided below and the results from the testbed are shown in the following chapter.

5.1 upc_all_broadcast

5.1.1 Push algorithm (myPUSH)

In the push based implementation of `MYSYNC` synchronization for upc_all_broadcast (`myPUSH`), each thread signals source thread that its ready to accept data and keep waiting for the exit signal from source thread. On the other hand, source thread loops on a busy wait until it serves all the threads. When it gets a signal from any of the threads, it sends the data to that particular thread and signals it to exit from the collective. In order to perform two successive calls to collectives with synchronization mode as `UPC_IN_MYSYNC | UPC_OUT_MYSYNC`, each thread maintains a collective call counter. Depending upon the counter value each thread decides which copy of two data structures to use for issuing synchronization signals in `MYSYNC` synchronization. This design

Collective In Synchronization	Collective Out Synchronization
UPC_IN_NOSYNC	UPC_OUT_NOSYNC
UPC_IN_NOSYNC	UPC_OUT_ALLSYNC
UPC_IN_NOSYNC	UPC_OUT_MYSYNC
UPC_IN_ALLSYNC	UPC_OUT_NOSYNC
UPC_IN_ALLSYNC	UPC_OUT_ALLSYNC
UPC_IN_ALLSYNC	UPC_OUT_MYSYNC
UPC_IN_MYSYNC	UPC_OUT_NOSYNC
UPC_IN_MYSYNC	UPC_OUT_ALLSYNC
UPC_IN_MYSYNC	UPC_OUT_MYSYNC

Figure 5.1: Synchronization types in UPC

avoids a race condition between consecutive collective calls. The pseudocode and data structure used for `MYSYNC` implementation is given below.

```
shared[THREADS] int upc_ready1[THREADS*THREADS];
shared[THREADS] int upc_ready2[THREADS*THREADS];
shared[THREADS] int upc_out1[THREADS*THREADS];
shared[THREADS] int upc_out2[THREADS*THREADS];


begin upc_all_broadcast(shared void *destination,
                        shared const void *source,
                        size_t nbytes,
                        upc_flag_t sync_mode)
source_thread = upc_threadof 'source' array
collective_count++;
count=THREADS;
if(collective_count is odd)
    if(MYTHREAD = source_thread)
    while(count)
```

```
        for i=0 to THREADS
          if(upc_ready1[MYTHREAD][i])
            upc_memcpy 'nbytes' from source into 'destination+i'
            upc_ready1[MYTHREAD][i]=0;
            upc_out1[i][MYTHREAD]=1;
            count--;
          end if
        end for
    end while
    else
    upc_ready1[MYTHREAD][source_thread]=1;
       while(1)
         if(upc_out1[MYTHREAD][source_thread])
           upc_out1[MYTHREAD][source_thread]=0;
           break;
         end if
       end while
    end ifelse
else
    /*Do the same for even collective call count using different set of
      data structures to avoid race condition*/
```

5.1.2 Pull algorithm (myPULL)

In the pull based implementation of MYSYNC synchronization for upc_all_broadcast (myPULL) source thread signals to each thread that it is ready to deliver data. It waits until it gets THREADS-1 exit signals. On the other hand, each thread loops on busy wait until it receives ready signal from source thread. Upon receiving ready signal, each thread pulls data from source thread and issues exit signal to source_thread. The Pseudocode used for MYSYNC implementation is given below.

```
begin upc_all_broadcast(shared void *destination,
                        shared const void *source,
                        size_t nbytes,
                        upc_flag_t sync_mode)
source_thread = upc_threadof 'source' array
collective_count++;
```

```
count=THREADS;
if(collective_count is odd)
    if(MYTHREAD = source_thread)
      for i=0 to THREADS
         upc_ready1[i][MYTHREAD]=1;
      end for
      while(count)
        for i=0 to THREADS
          if(upc_out1[MYTHREAD][i])
            upc_out1[MYTHREAD][i]=0;
            count--;
          endif
        end for
      end while
    else
      while(1)
        if(upc_ready1[MYTHREAD][source_thread])
          upc_memcpy 'nbytes' from source into 'destination+i'
          upc_ready1[MYTHREAD][source_thread]=0;
          break;
        end if
      end while
      upc_out1[source_thread][MYTHREAD]=1;
      end ifelse
else
    /*Do the same for even collective call count using different set of
      data structures to avoid race condition*/
```

5.1.3 GM based push algorithm (myGMTU)

The `myGMTU` algorithm makes use of a tree based broadcast algorithm which uses a fixed tree structure as shown in Figure 5.2. In Figure 5.2 thread 0 is the source thread. If thread 2 is the source thread, then the data transfer takes place as shown in Figure 5.3. Here, we swap source thread node with thread 0 node in the tree structure in Figure 5.2 and use the modified tree to transfer the data.

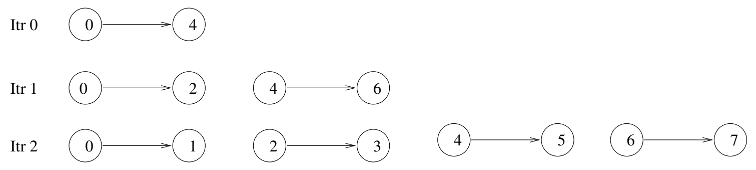


Figure 5.2: Static tree Broadcast1

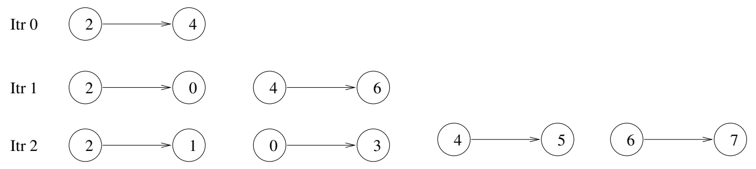


Figure 5.3: Static tree Broadcast2

Implementation of `MYSYNC` synchronization for this push based algorithm takes place in four steps.

1. Each thread signals its parent that it has arrived at the collective call.
2. Each thread determines the child threads to which it is going to transfer data.
3. Each thread waits for the data signal from its parent thread
4. When any child thread is ready to receive data, data is transferred

5.2 upc_all_scatter

The implementation of the myPULL and myPUSH algorithms for `upc_all_scatter` is very similar to the one explained for `upc_all_broadcast`. For myGMTU algorithm, instead of using the tree based structure we transfer data as in the myPUSH implementation.

5.3 upc_all_gather

In the push based implementation of `MYSYNC` in `upc_all_gather` destination thread signals each thread that it has arrived at the collective and wait for an exit signal from all other threads. On the other hand, all the `THREADS`$-$`1` threads wait for an arrival signal from the destination thread. Upon receiving the arrival signal each thread transfers data to the destination thread and sends an exit signal to it. For the myGMTU algorithm the same synchronization mechanism is used.

In the pull based implementation each thread reports its arrival to the destination thread and waits for an exit signal from destination thread. Each destination thread waits and then pulls data from the source thread as soon as it receives an arrival signal. The destination thread then send an exit signal to source thread. After pulling that from all threads destination thread exits the collective call.

5.4 upc_all_gather_all

For `upc_all_gather_all` collective function, pull based as well as push based implementations are same because each thread has to wait for all other threads. To implement `MYSYNC` synchronization each thread signals its arrival to all other threads and waits for arrival signal from all other threads. As soon as it receives an arrival signal from another thread it either pushes/pulls data to/from the other thread depending upon type of algorithm.

5.5 upc_all_exchange

`upc_all_exchange` collective function moves data in a pattern very similar to the collective `upc_all_gather_all`. The only difference is the way data is transferred from source array to destination array. Thus, the synchronization mechanism used is same as `upc_all_gather_all`.

5.6 upc_all_permute

In the push based implementation each thread sends signal to the thread with id equal to `permute[MYTHREAD]` about its arrival and finds its partner thread by searching its thread id in the `permute` array. Upon receiving signal from its partner each thread transfers data and send an exit signal. As soon as a thread receives an exit signal from its partner thread and thread with id `permute[ MYTHREAD]` it leaves the collective call. The same synchronization mechanism is used for the myGMTU algorithm.

In the pull based implementation each thread send an arrival signal to its partner thread. Upon receiving signal from `permute[ MYTHREAD]` each thread transfers data to it and send an exit signal. As soon as each thread receives an exit signal from partner thread and from thread with id `permute[ MYTHREAD]` it leaves the collective call.

Chapter 6

Performance Evaluation

6.1 Testbed

Standard benchmarks, like the Pallas MPI benchmarks (PMB), were not specifically constructed to measure collective communication time for different collective libraries. In these benchmark, computational time dominates the collective time by several orders of magnitude. Therefore, we are going to use a testbed developed by Alok Mishra at MTU that was specifically developed to compare collective communication times of different collective libraries. This testbed allows us to set computational time as per requirement. The term *collective time* is the maximum time spent by any thread in the collective call. Computational time can be defined as the time spent by any thread in computing something between two consecutive collective calls.

The testbed interleaves computation and collective communication, to provide comparisons close to real world applications [4]. In order to compare the `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC` synchronization mode Mishra used to set computational time slightly higher than the collective communication time, to ensure that all threads have completed one round of collective communication before entering the next round. In our case, we want to compare collective communication time for the `UPC_IN_MYSYNC | UPC_OUT_MYSYNC` synchronization mode and we came up with some modifications to the testbed which are explained in the following subsection.

The testbed conducts a warm-up routine with a small number of collective calls before conducting the actual test runs. This allows us to obtain the raw collective time with no computation. The collective time measured is then used to determine how much computation should be done between two successive collective calls. All of the computation is local and there are no remote operations during the compute phase. In Mishra's testbed the computation time was set to twice that of the collective time to quiet the network before measuring the collective time again. In order to calculate the collective time, the testbed trials t is set to be 100 and the number of threads is assumed to be T. Each thread determines the time taken by collective call for all the testbed trials. Let t_{ij} be the time the collective call took on thread i for trial j. Each thread i then determines the average time taken by the collective calls in the testbed trials as $a_i = \frac{1}{t}\sum_{j=1}^{t} t_{ij}$. At the end, thread 0 calculates $\max_{i=0}^{T-1} a_i$, which is the maximum of average collective call time in testbed trials on all threads. This maximum time is reported as the collective time. In order to calculate the overall time testbed computes the slowest thread i for which a_i is maximum. Testbed then reports time taken by the slowest thread to execute the entire set of trials as the overall time for the testbed.

6.1.1 Modifications in Testbed

During the study of the testbed we found that there is need to modify the testbed's approach of *collective* and *overall* time measurement. All the modifications mentioned below are incorporated to the testbed algorithm to obtain the results.

- In order to more consistently measure the difference in overall time due to `MYSYNC` synchronization compared to `ALLSYNC` synchronization, we decided to set the computational time to be twice that of the collective time of the slowest algorithm. With this arrangement, for a given message length each thread does the same amount of computation irrespective of the algorithm.

- In order to determine collective time, each thread reports its collective time to thread 0 for each testbed trial. Thread 0 determines the maximum of the reported timings for each trial. The Maximum time collective call took for trial j on all threads is $m_j = \max_{i=0}^{T-1}\{t_{ij}\}$. At the end of the testbed trials, thread 0 calculates the collective time to be $c = \frac{1}{t}\sum_{j=1}^{t} m_j$ which is the average of the maximum times on each trial.

- The overall time is the maximum time taken by any thread to complete all the testbed trials.

6.1.2 Testbed parameters

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.

- `Hardware platform`

  We tested our library on a 15 node Linux/Myrinet cluster. The cluster is also connected by a fast ethernet connection. We used the `MPICH-GM` distribution of the `MuPC` runtime system for UPC.

- `Algorithms`

  The testbed can be compiled by linking different collective libraries and comparing one implementation with another. We compared the performance of the following implementations.

  - `mPULL` - Reference PULL based implementation of MYSYNC synchronization
  - `mPUSH` - Reference PUSH based implementation of MYSYNC synchronization
  - `myPULL` - PULL based (UPC) implementation of MYSYNC synchronization
  - `myPUSH` - PULL based (UPC) implementation of MYSYNC synchronization
  - `GMTU` - PUSH based (GM level) implementation of MYSYNC synchronization
  - `myGMTU` - PUSH based (GM level) implementation of MYSYNC synchronization

- `Collective Operation`

  The testbed is designed to measure the performance of the following collective:

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

- `Number of threads involved`

  The number of processes, or UPC threads, varies from 2 to 15. (Currently one node on lionel is not functioning and hence we can use maximum 15 threads. Run times were measured for 2 8 and 15 threads.

- `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 *NO*, are used in combination to specify the synchronization before and after data transfer within a collective. We set the synchronization mode for the collective testbed to be `UPC_IN_MYSYNC | UPC_OUT_MYSYNC`.

- `Message Length`

  Performance was measured for message lengths from 8 bytes to 16K bytes.

- `Number of testbed runs`

  The testbed runs 100 identical trials to determine collective time and overall time.

- `Computational coefficient`

  This parameter is used to compute computational time from the warm-up collective time of slowest algorithm which is assumed to be the `mPUSH` implementation of each collective. Computational time is set to twice that of computational coefficient.

6.2 Performance analysis

In the implementation of `MYSYNC` in broadcast and scatter, either the source thread reports its arrival at the collective call to all other threads or each thread reports its arrival to the source thread. In the case of the gather collective operation each thread reports to the destination thread about its arrival or the destination thread reports its arrival to all other threads. In all cases this results in `THREADS`-1 messages. In the gather_all and exchange collectives each thread reports to all other threads about its arrival at the collective call resulting in `THREADS*(THREADS` -1`)` messages. For the permute operation, each thread reports to its partner thread about its arrival resulting in `THREADS` messages. Thus, there are $2 * (\texttt{THREADS} - 1)$ messages required in the implementation of `MYSYNC` for broadcast, scatter and gather; $2 * \texttt{THREADS}$ messages for the permute collective operation and $2 * \texttt{THREADS} * (\texttt{THREADS} - 1)$ messages are required for the gather_all and exchange collective operations.

UPC_COLLECTIVES	ALLSYNC	MYSYNC
upc_all_broadcast	4*(THREADS−1)	2*(THREADS−1)
upc_all_scatter	4*(THREADS−1)	2*(THREADS−1)
upc_all_gather	4*(THREADS−1)	2*(THREADS−1)
upc_all_gather_all	4*(THREADS−1)	2*THREADS*(THREADS−1)
upc_all_exchange	4*(THREADS−1)	2*THREADS*(THREADS−1)
upc_all_permute	4*(THREADS−1)	2*(THREADS−1)

Figure 6.1: Message overhead in `ALLSYNC`,`MYSYNC` synchronization

On the other hand, `upc_barrier`, which is used to implement the `ALLSYNC` mode, causes $2 * (\texttt{THREADS} - 1)$ messages. As `upc_barrier` is used twice, when entering and leaving the collective, the total messages required to implement `ALLSYNC` mode are $4 * (\texttt{THREADS} - 1)$.

Table in Figure 6.1 summarizes the message overhead for `ALLSYNC` and `MYSYNC` in each of the six relocalization collectives.

The second parameter which has impact on the performance of collectives is number of communication steps involved in synchronization mechanism. Figure 6.2 gives the number of communication steps, for the MYSYNC and ALLSYNC modes, for the 6 relocalization collectives.

UPC_COLLECTIVES	ALLSYNC	MYSYNC
upc_all_broadcast	8*[log (THREADS)−1]	2*(THREADS−1)
upc_all_scatter	8*[log (THREADS)−1]	2*(THREADS−1)
upc_all_gather	8*[log (THREADS)−1]	2*(THREADS−1)
upc_all_gather_all	8*[log (THREADS)−1]	2*(THREADS−1)
upc_all_exchange	8*[log (THREADS)−1]	2*(THREADS−1)
upc_all_permute	8*[log (THREADS)−1]	2

Figure 6.2: Number of communication steps in `ALLSYNC` and `MYSYNC`

Other than the number of communication steps and message overhead performance is affected by parallelism in data transfer and synchronization with other threads. `MYSYNC` mode can make progress on data transfer if some threads arrive late at the collective call. On the other hand, `upc_barrier` used to implement `ALLSYNC`, do not allow transfer of data until all the threads are synchronized. Thus the waiting time of source thread is reduced due to parallelism in `MYSYNC` which leads to better performance. Reduced waiting time of all other threads in `MYSYNC`, as compared to `ALLSYNC` results in overall lesser execution time of an application.

6.2.1 Result:upc_all_broadcast

The results for 2 and 8 threads, comparing the UPC level pull-based implementation of `MYSYNC` against `ALLSYNC` and myPUSH and myGMTU implementations against reference push implementation and Mishra's GMTU implementation are shown below.

6.2.2 Result:upc_all_scatter

The results for 2 and 8 threads, comparing the UPC level pull-based implementation of `MYSYNC` against `ALLSYNC` and myPUSH and myGMTU implementations against reference push implementation and Mishra's GMTU implementation are shown below.

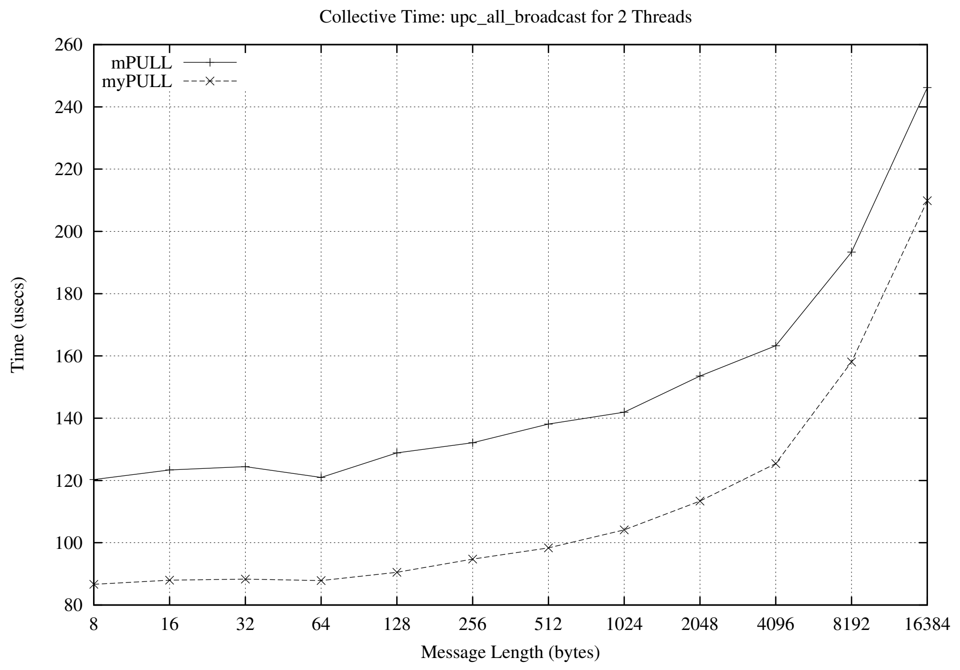


Figure 6.3: Collective Time: `upc_all_broadcast`, 2 Threads, Pull implementation

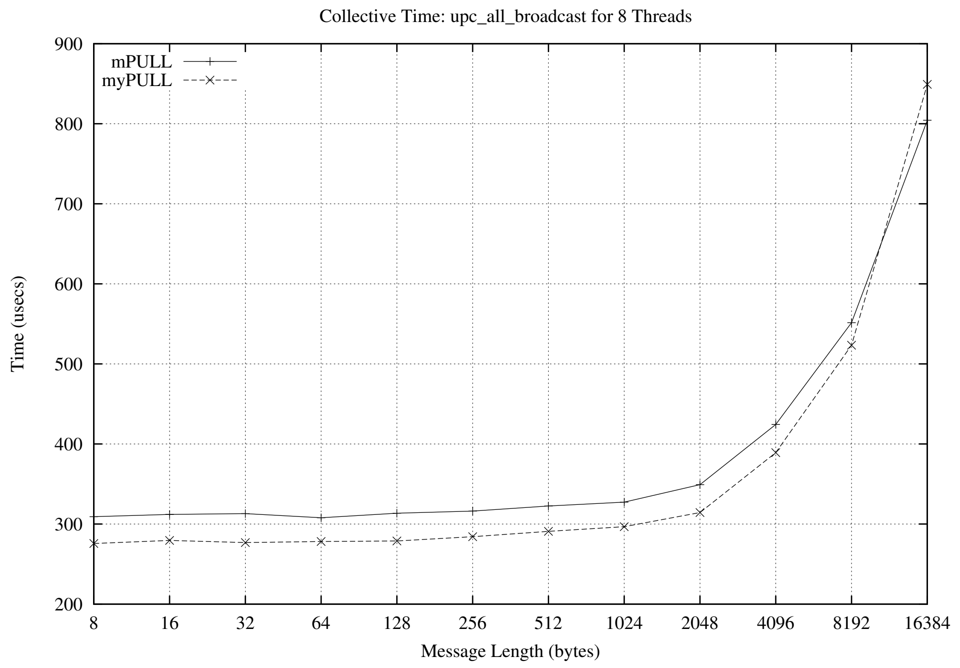


Figure 6.4: Collective Time: `upc_all_broadcast`, 8 Threads, Pull implementation

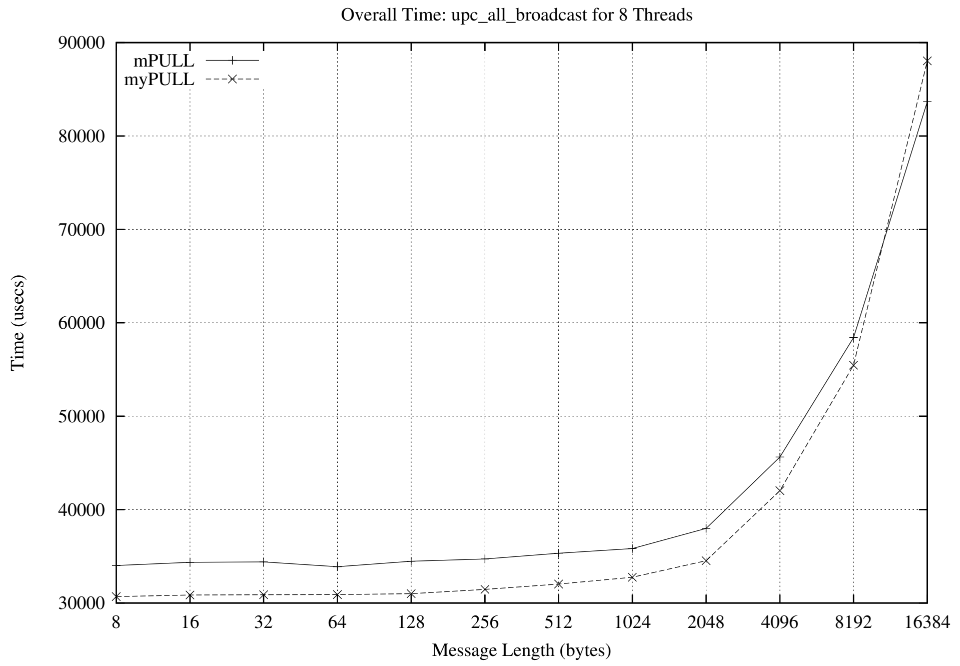


Figure 6.5: Overall Time: `upc_all_broadcast`, 8 Threads, Pull implementation

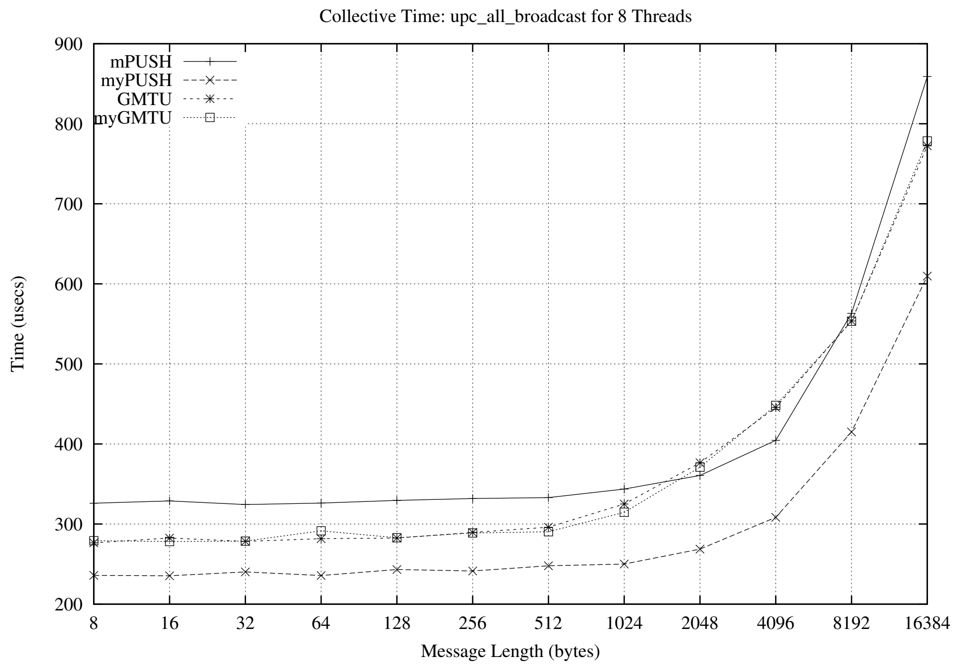


Figure 6.6: Collective Time: `upc_all_broadcast`, 8 Threads, Push implementations

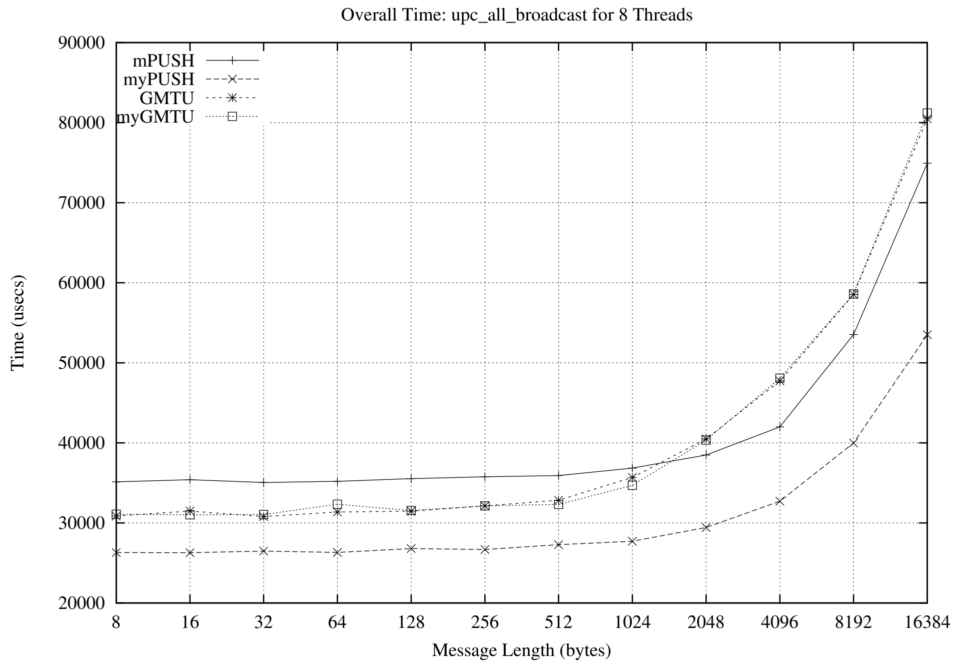


Figure 6.7: Overall Time: `upc_all_broadcast`, 8 Threads, Push implementations

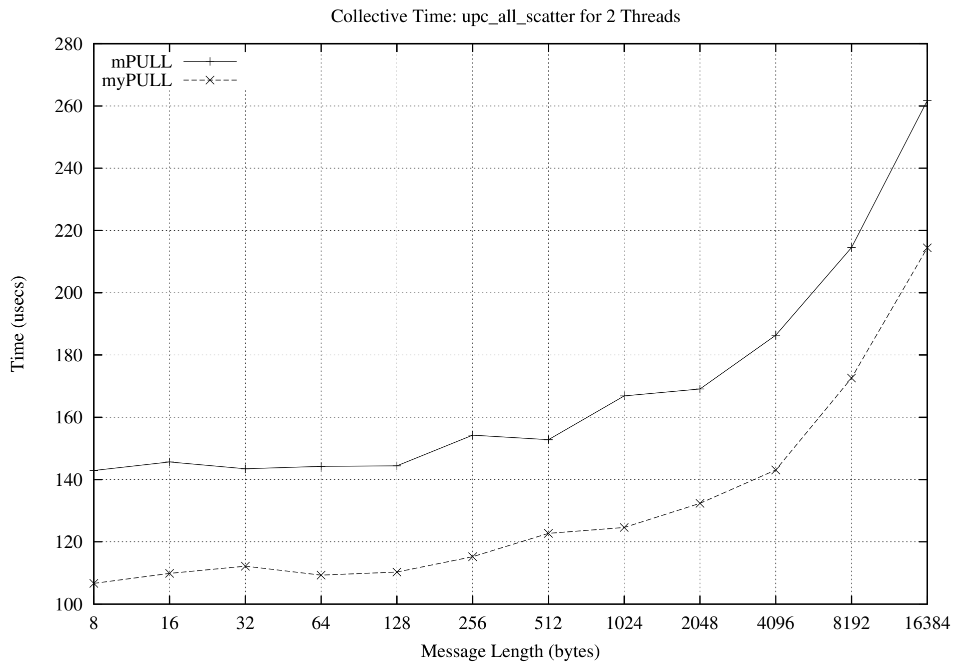


Figure 6.8: Collective Time: `upc_all_scatter`, 2 Threads, Pull implementation

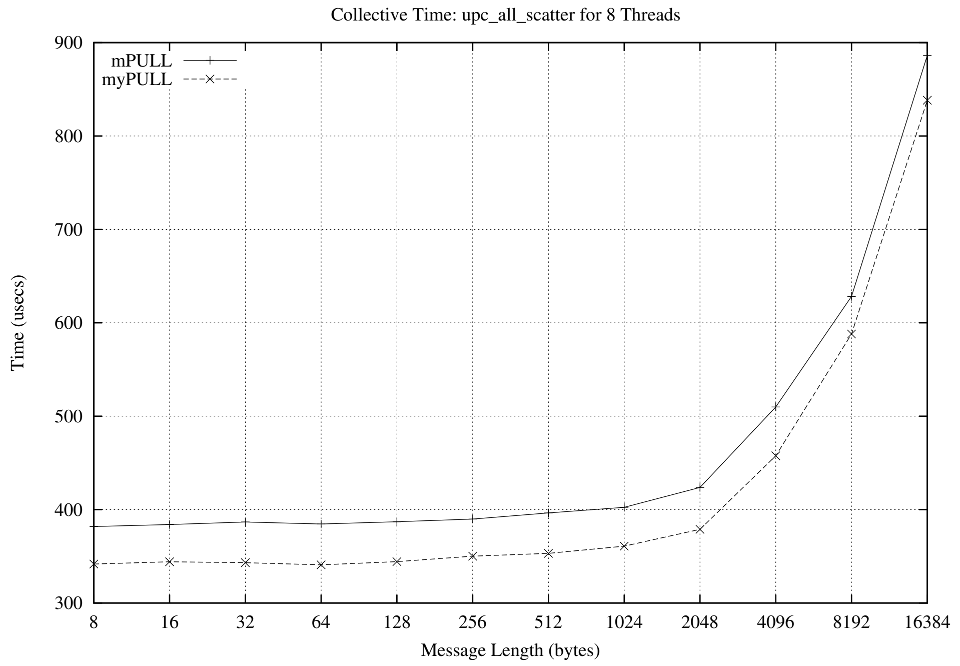


Figure 6.9: Collective Time: `upc_all_scatter`, 8 Threads, Pull implementation

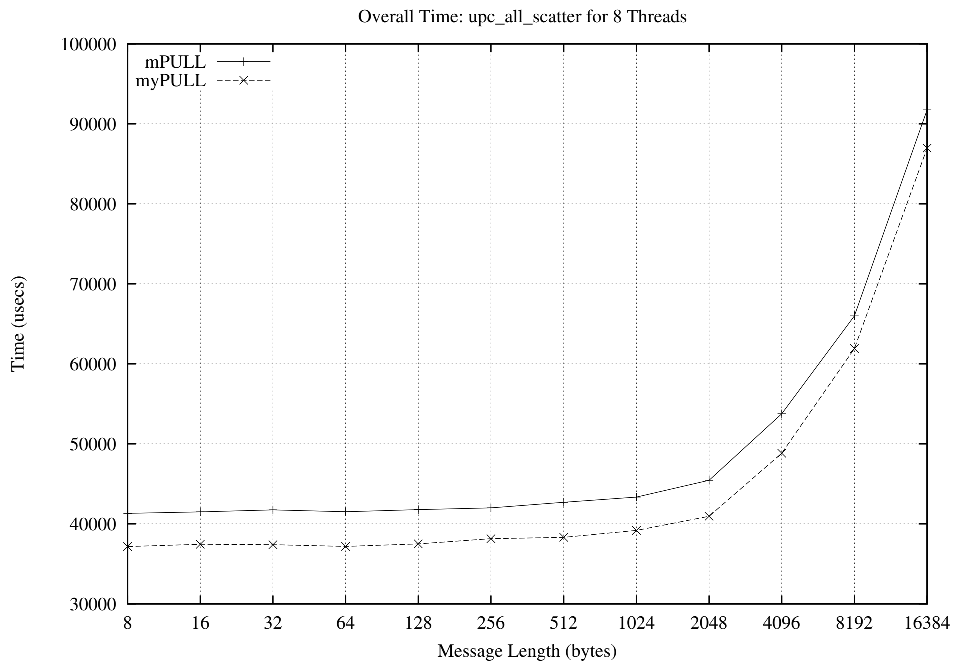


Figure 6.10: Overall Time: `upc_all_scatter`, 8 Threads, Pull implementation

6.2.3 Result:upc_all_gather

The results for 2 and 8 threads, comparing the UPC level pull-based implementation of `MYSYNC` against `ALLSYNC` and myPUSH and myGMTU implementations against reference push implementation and Mishra's GMTU implementation are shown below.

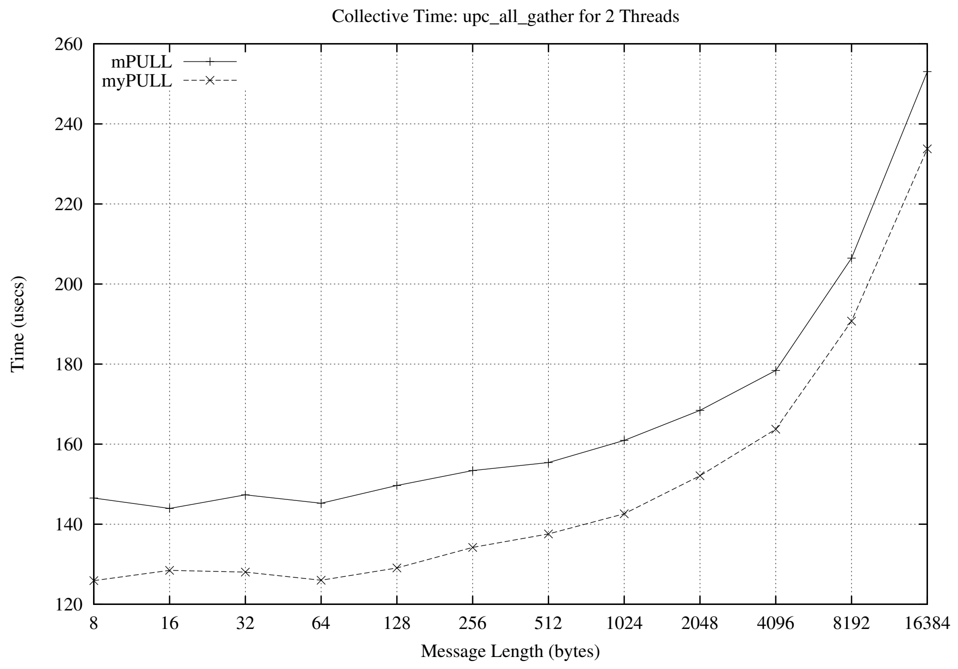


Figure 6.11: Collective Time: `upc_all_gather`, 2 Threads, Pull implementation

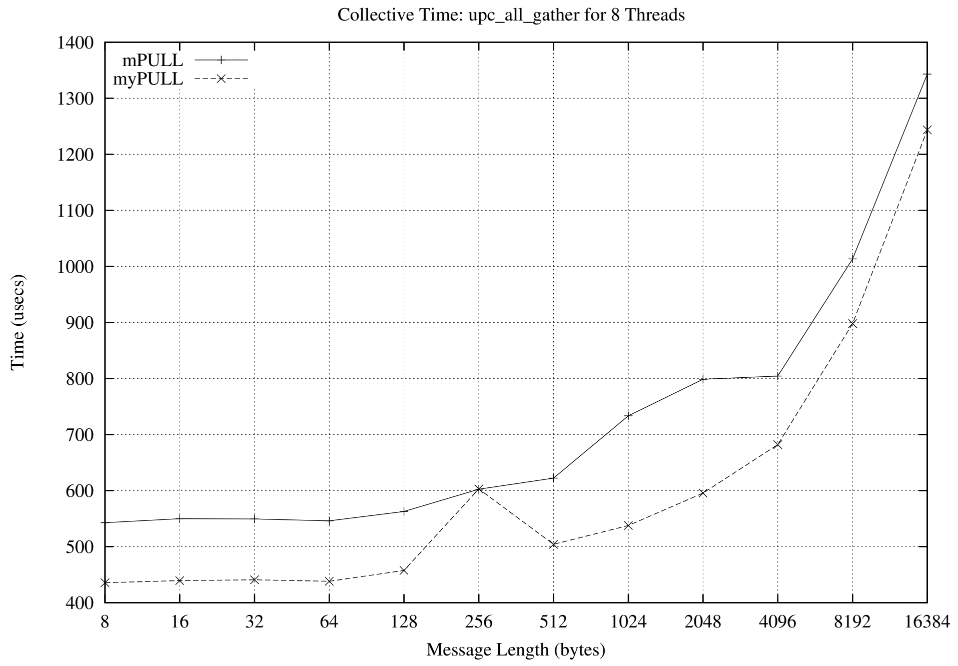


Figure 6.12: Collective Time: `upc_all_gather`, 8 Threads, Pull implementation

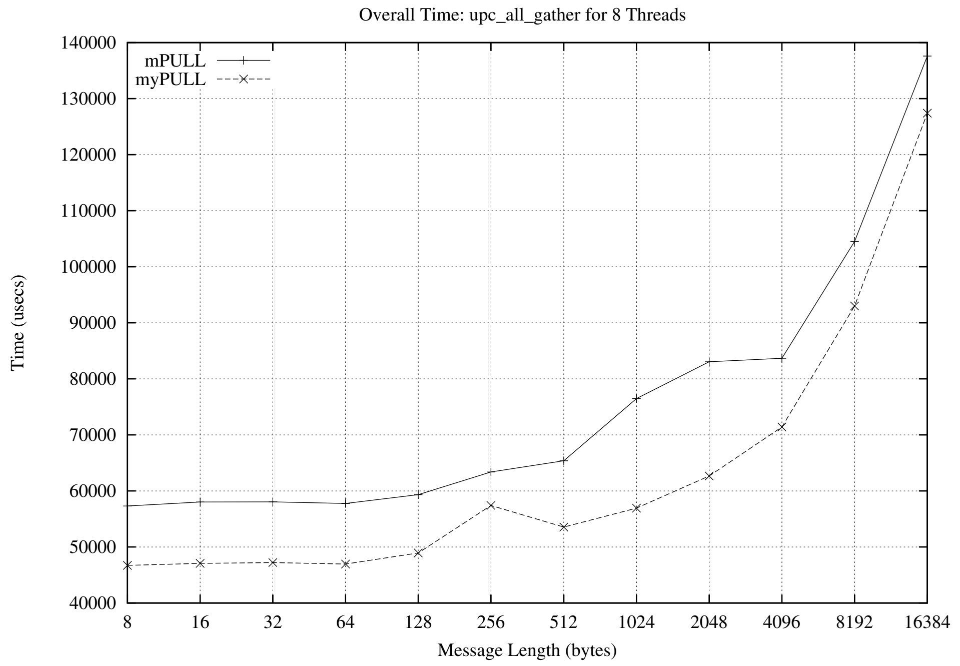


Figure 6.13: Overall Time: `upc_all_gather`, 8 Threads, Pull implementation

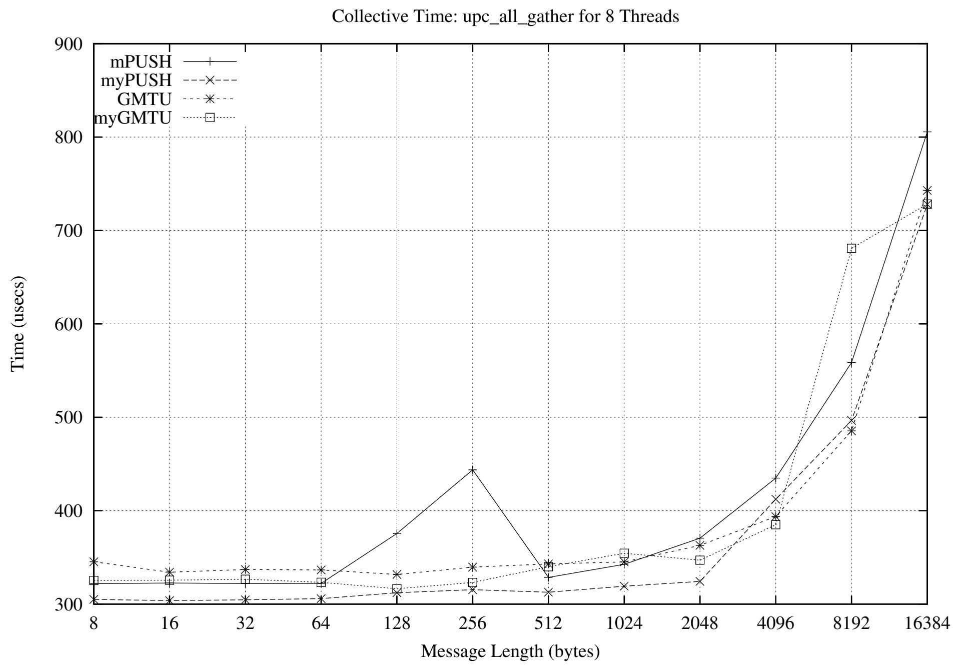


Figure 6.14: Collective Time: `upc_all_gather`, 8 Threads, Push implementations

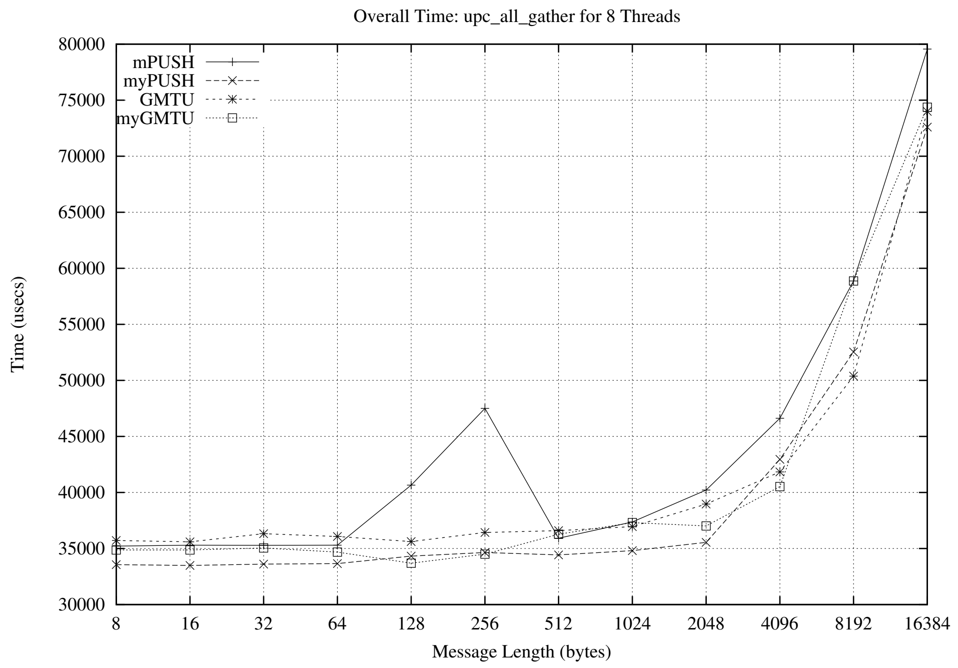


Figure 6.15: Overall Time: `upc_all_gather`, 8 Threads, Push implementations

6.2.4 Result:upc_all_gather_all

The results for 2 and 8 threads, comparing the UPC level pull-based implementation of `MYSYNC` against `ALLSYNC` and myPUSH and myGMTU implementations against reference push implementation and Mishra's GMTU implementation are shown below.

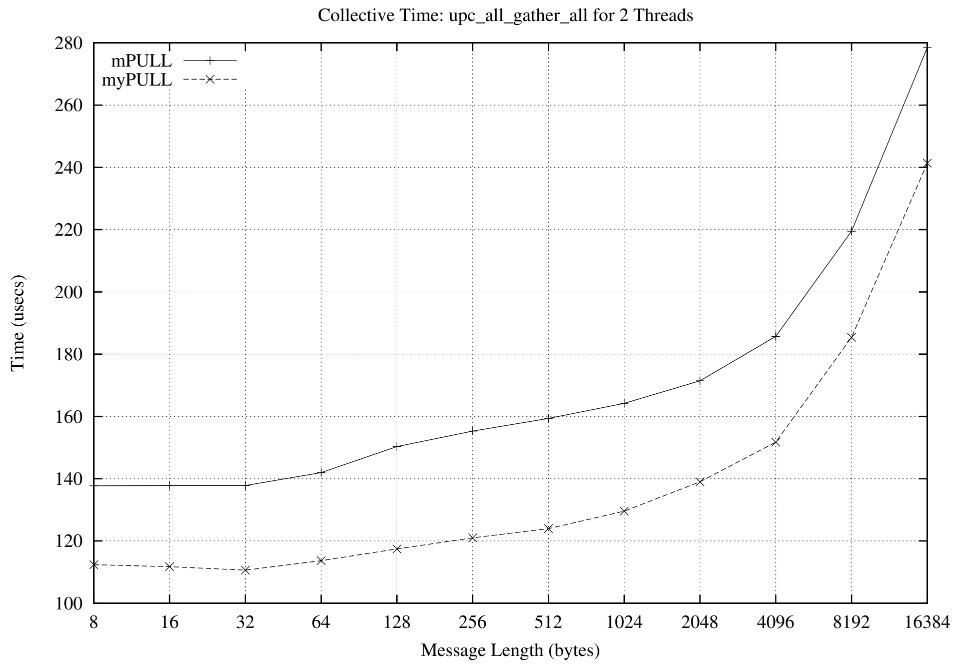


Figure 6.16: Collective Time: `upc_all_gather_all`, 2 Threads, Pull implementation

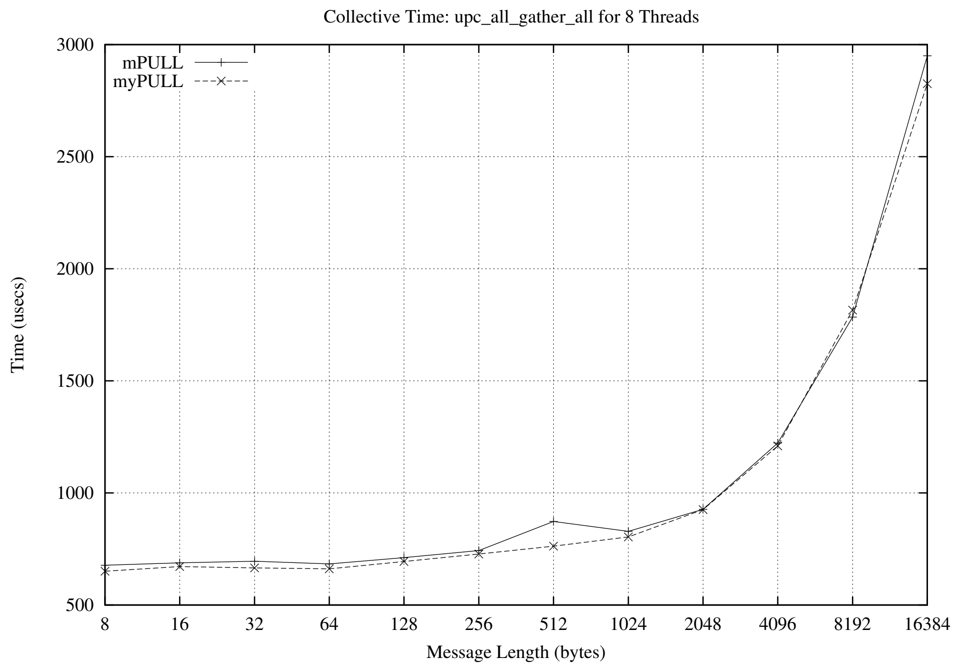


Figure 6.17: Collective Time: `upc_all_gather_all`, 8 Threads, Pull implementation

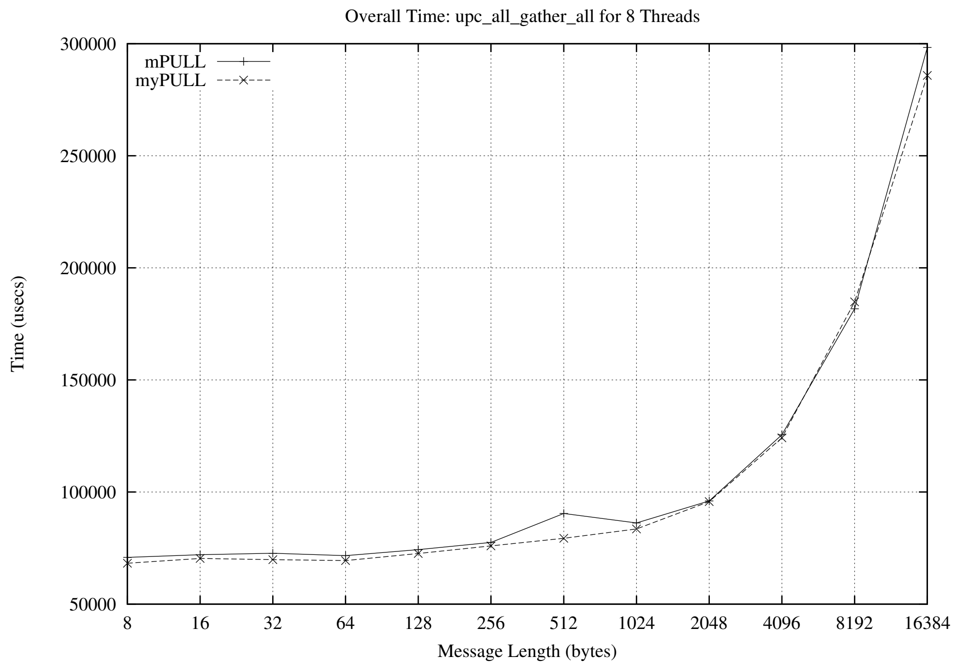


Figure 6.18: Overall Time: `upc_all_gather_all`, 8 Threads, Pull implementation

6.2.5 Result:upc_all_exchange

The results for 2 and 8 threads, comparing the UPC level pull-based implementation of `MYSYNC` against `ALLSYNC` and myPUSH and myGMTU implementations against reference push implementation and Mishra's GMTU implementation are shown below.

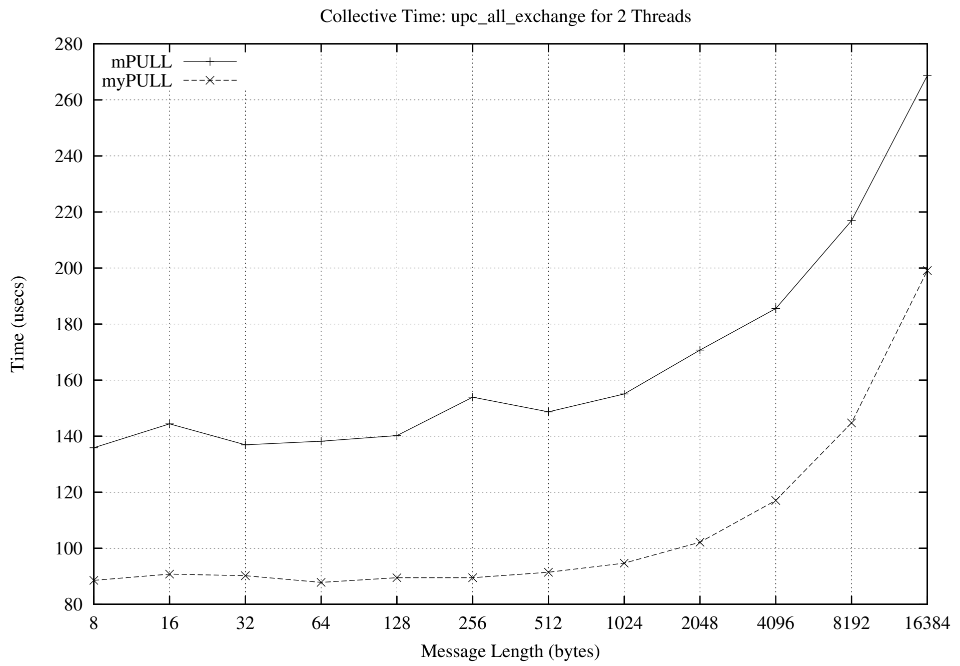


Figure 6.19: Collective Time: `upc_all_exchange`, 2 Threads, Pull implementation

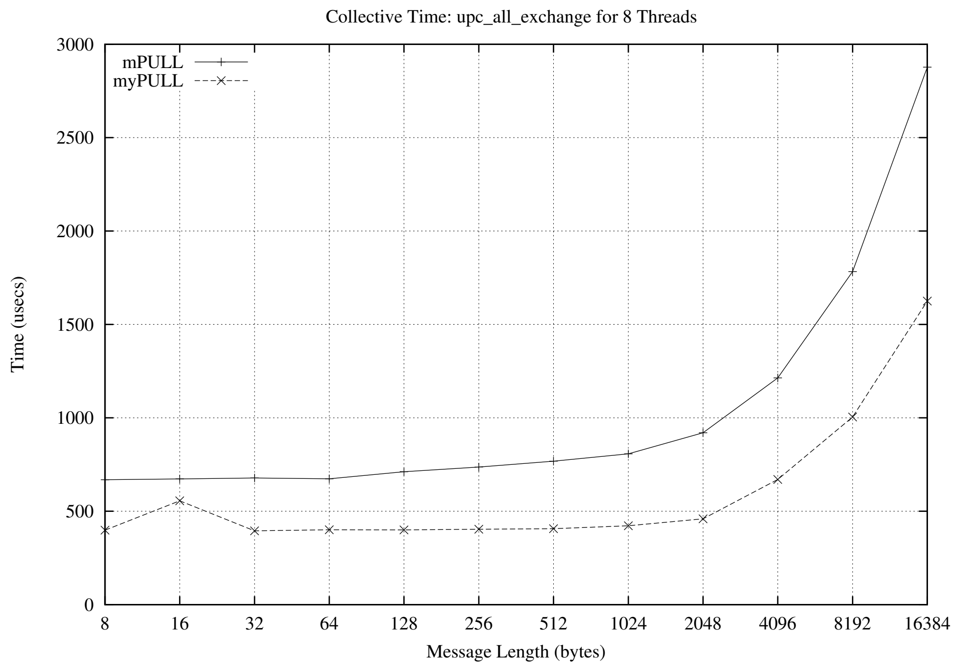


Figure 6.20: Collective Time: `upc_all_exchange`, 8 Threads, Pull implementation

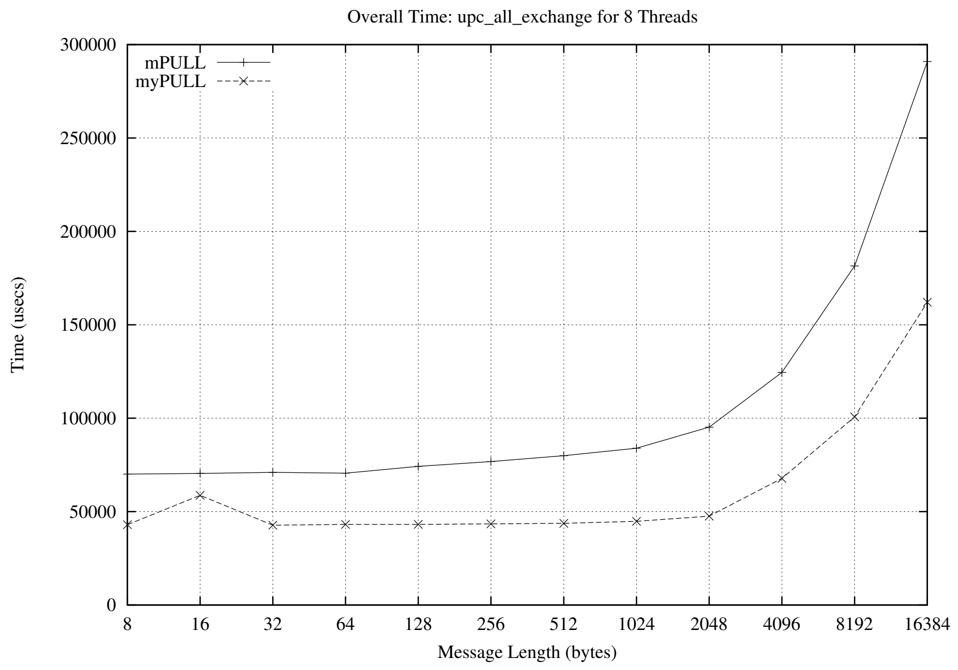


Figure 6.21: Overall Time: `upc_all_exchange`, 8 Threads, Pull implementation

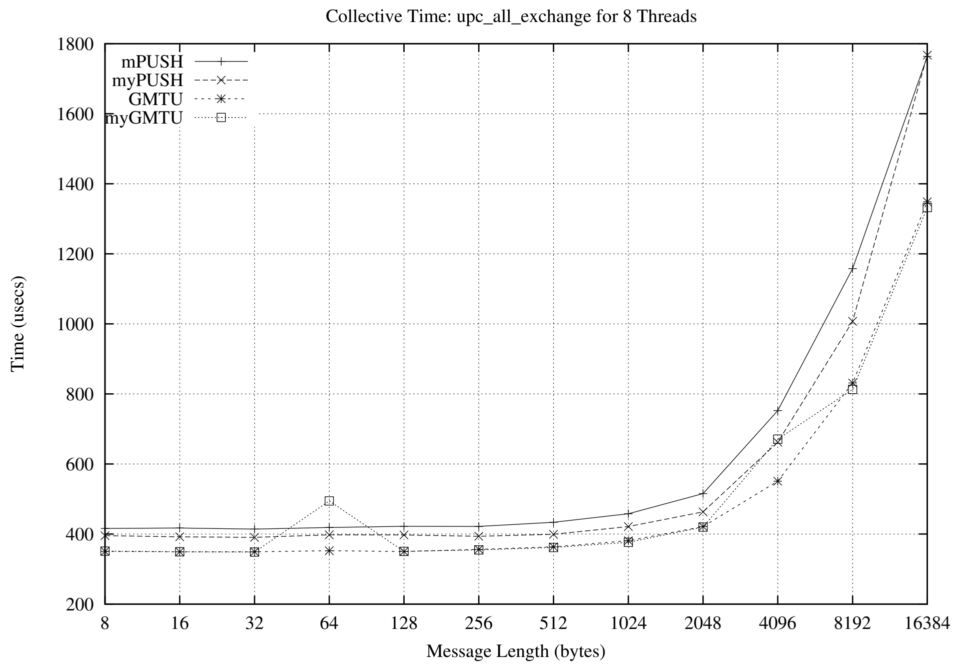


Figure 6.22: Collective Time: `upc_all_exchange`, 8 Threads, Push implementations

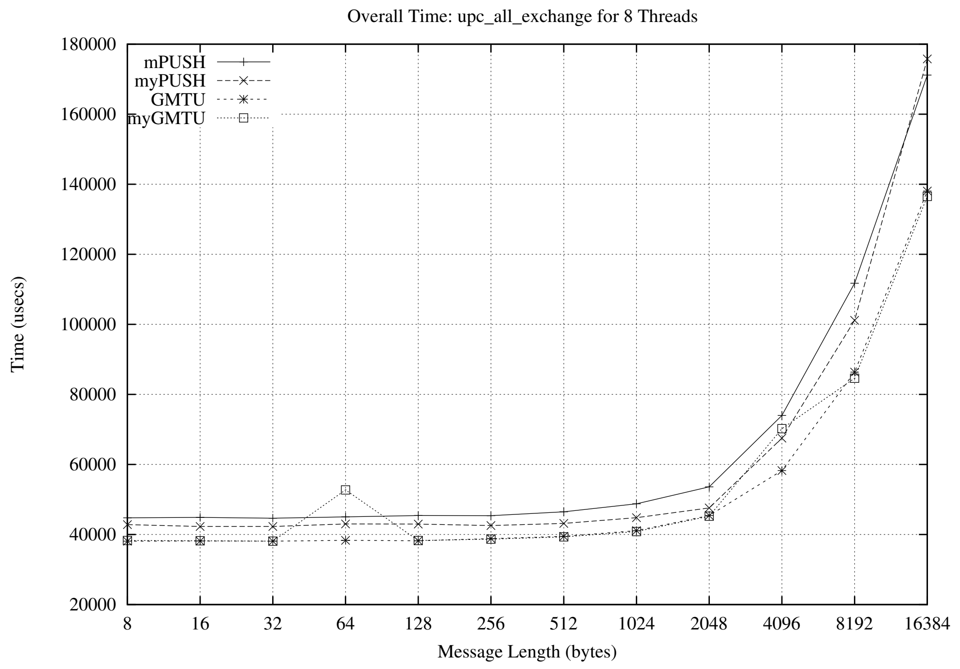


Figure 6.23: Overall Time: `upc_all_exchange`, 8 Threads, Push implementations

6.2.6 Result:upc_all_permute

The results for 2 and 8 threads, comparing the UPC level pull-based implementation of `MYSYNC` against `ALLSYNC` and myPUSH and myGMTU implementations against reference push implementation and Mishra's GMTU implementation are shown below.

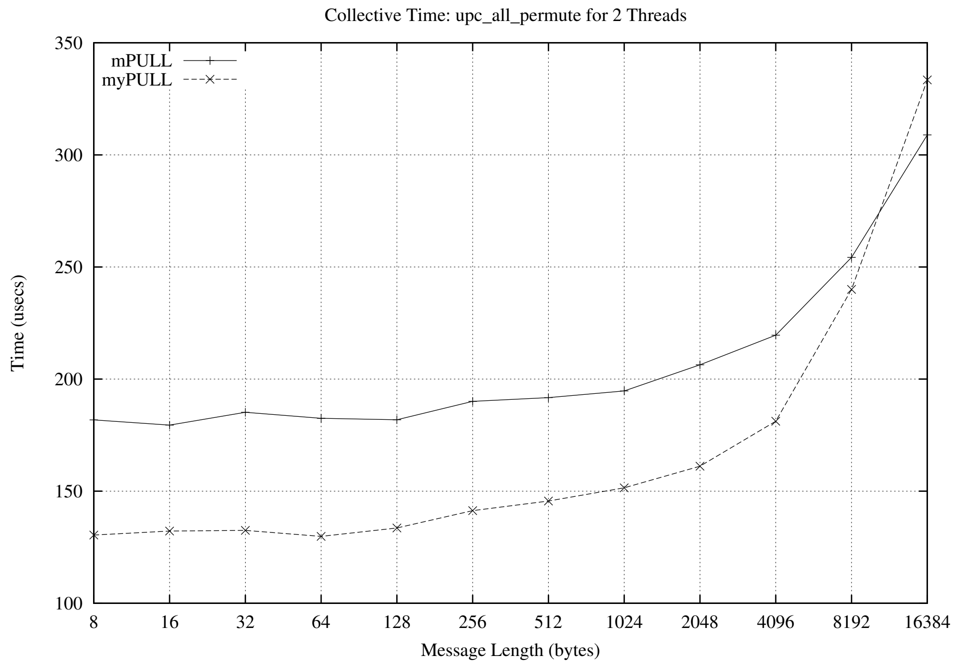


Figure 6.24: Collective Time: `upc_all_permute`, 2 Threads, Pull implementation

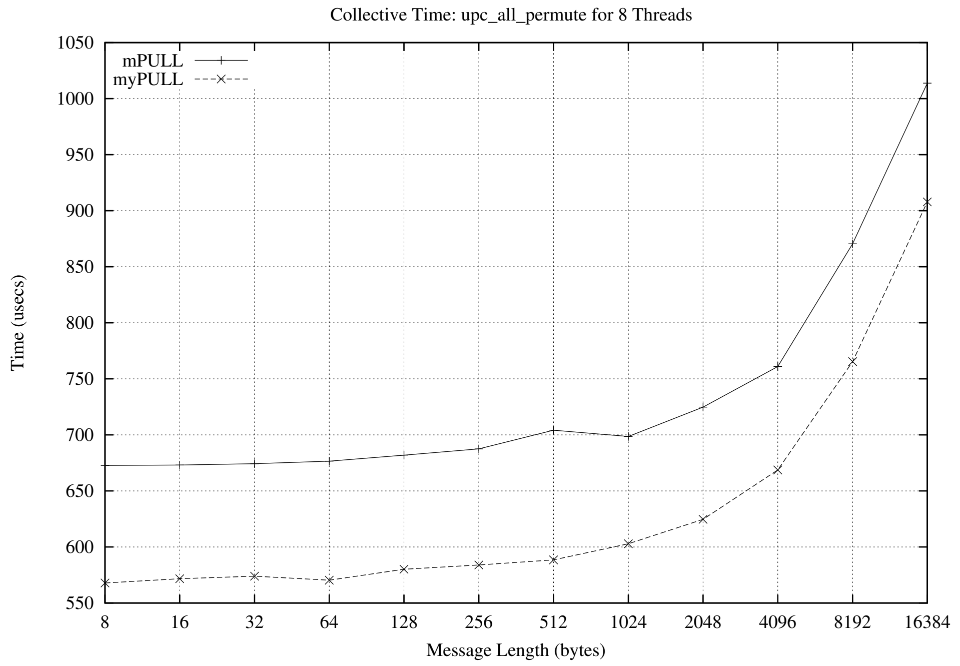


Figure 6.25: Collective Time: `upc_all_permute`, 8 Threads, Pull implementation

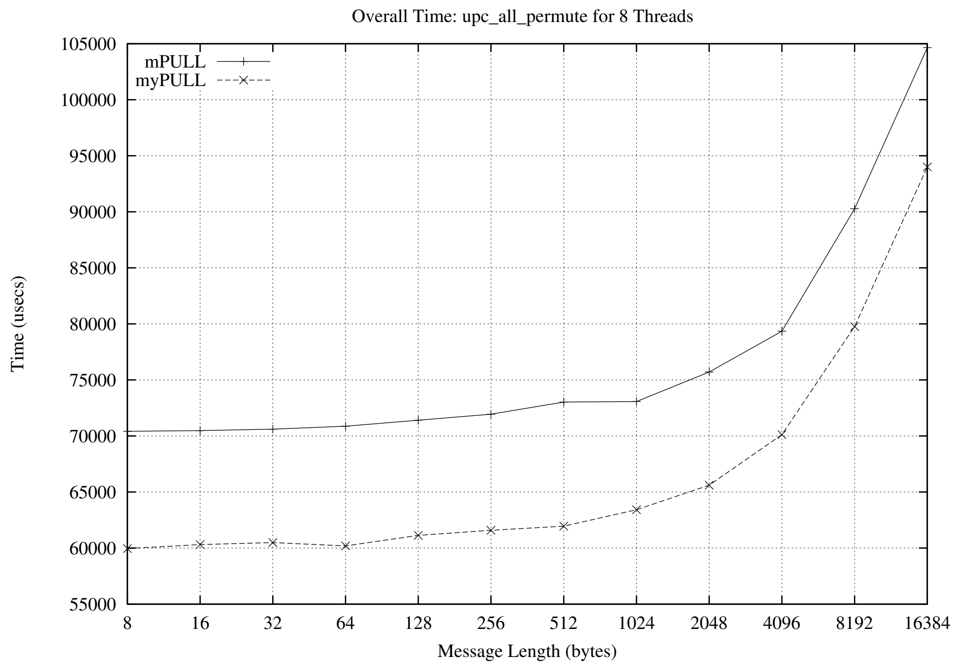


Figure 6.26: Overall Time: `upc_all_permute`, 8 Threads, Pull implementation

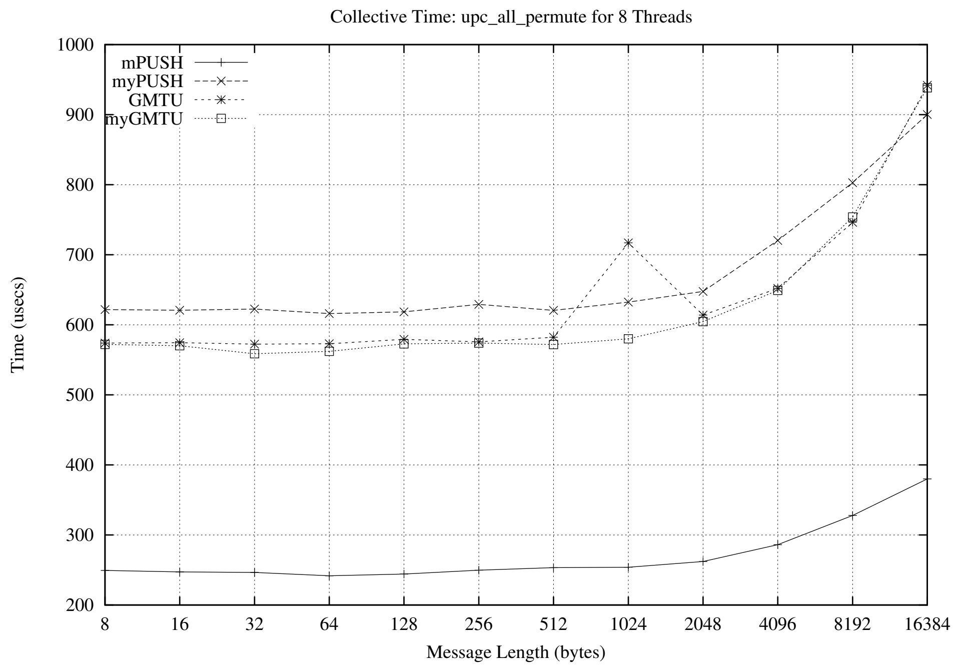


Figure 6.27: Collective Time: `upc_all_permute`, 8 Threads, Push implementations

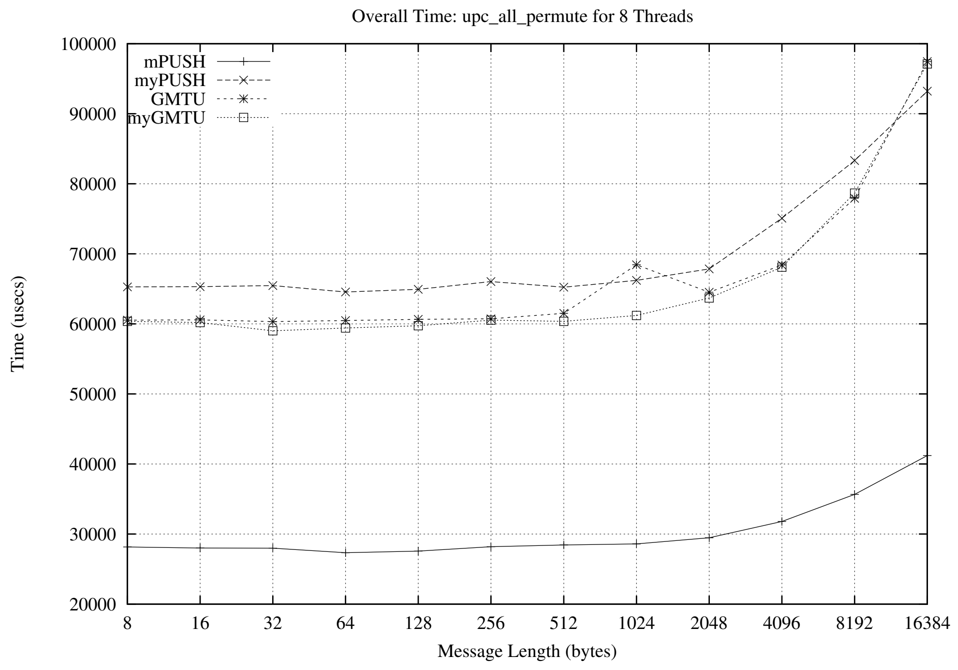


Figure 6.28: Overall Time: `upc_all_broadcast`, 8 Threads, Push implementations

6.3 Effect of number of threads on performance

The following diagrams illustrates the collective time and overall time for `upc_all_broadcast` on 15 threads. We compared the results for `upc_all_broadcast` on 2 and 8 threads, shown in Figure 6.3, 6.4 and 6.5 respectively, with the results obtained on 15 threads in Figures 6.29 and 6.30.

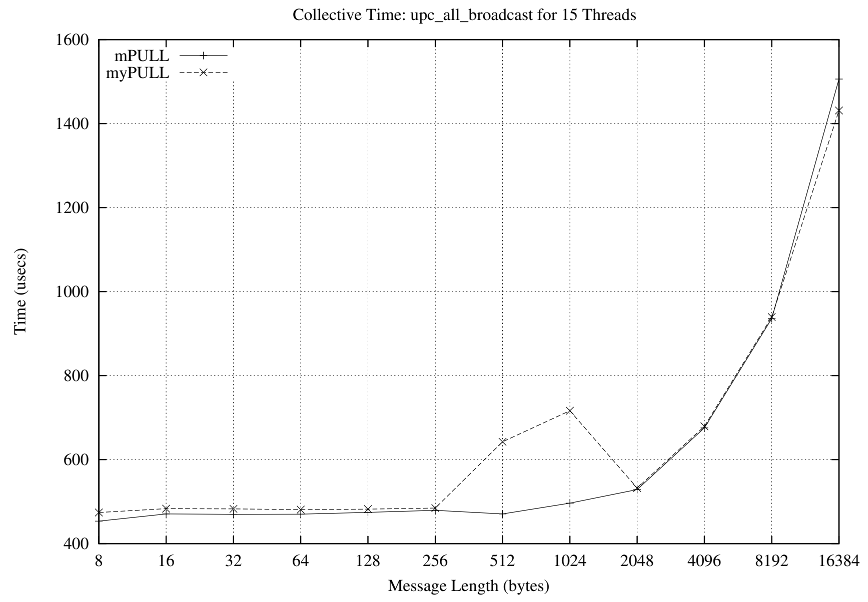


Figure 6.29: Collective Time: `upc_all_permute`, 15 Threads, Pull implementations

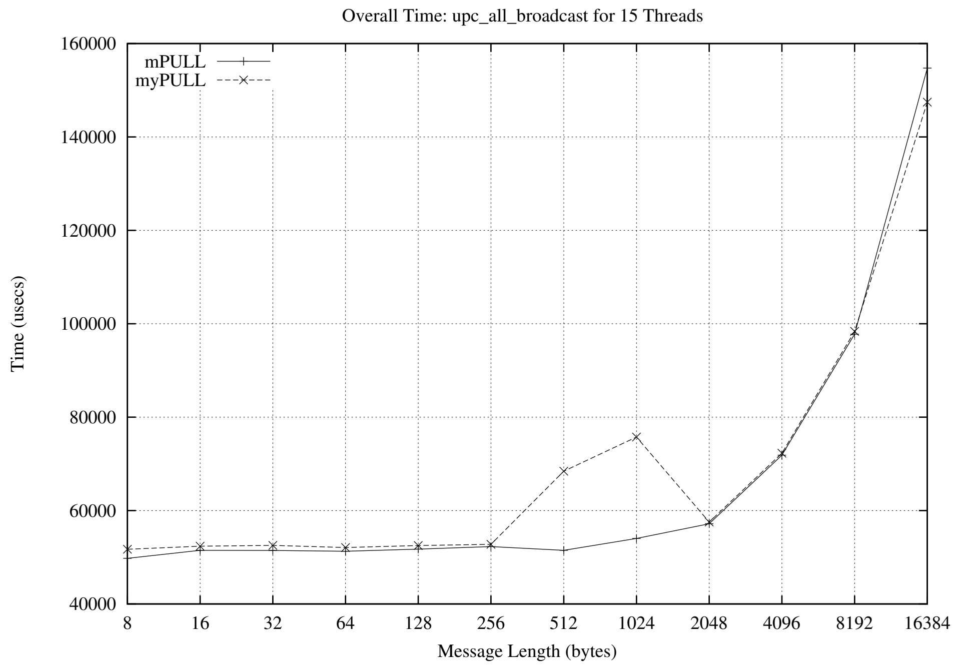


Figure 6.30: Overall Time: `upc_all_broadcast`, 15 Threads, Pull implementations

We noticed that the performance improvement due to `MYSYNC` mode reduces with increase in number of threads . This is because the number of communication steps are proportional to $log($`THREADS`$)$ in `ALLSYNC` mode whereas in the `MYSYNC` mode communication steps are proportional to number of `THREADS`. Thus increasing the number of threads, increases the synchronization time required in `MYSYNC` mode. This leads to increase in collective time and overall time.

6.4 Conclusion

The results that we got from the testbed indicate that the collective time increases with increase in the number of threads. Increase in the collective time leads to increase in the overall time. Also, the performance improvement that we got in `MYSYNC` mode than `ALLSYNC` mode is independent of the message length. This is because the performance improvement is due to collective communication steps and the parallelism in synchronization and data transfer. We also noticed that in the push implementation of `Permute`, the `ALLSYNC` synchronization performs better than the `MYSYNC` synchronization mode. This is because in the `MYSYNC` mode each thread finds the partner thread by accessing data from shared memory to which thread does not have affinity.

Chapter 7

Conclusion

The UPC-level library developed in this project implements the MYSYNC synchronization mode along with ALLSYNC and NOSYNC reference implementation, providing users with all possible 9 synchronization modes for collectives. Users can also use the GM version of the library which integrates the MYSYNC implementation with Mishra's implementations of ALLSYNC and NOSYNC. The MYSYNC implementation makes use of pairwise synchronization of threads and thus each thread avoids waiting for all other threads with which it does not need to communicate.

As per the analysis, the collective time mainly consists of communication steps in synchronization, message overhead due to synchronization and data transfer time. In the MYSYNC implementation message overhead due to synchronization is less than in ALLSYNC. But in ALLSYNC mode, synchronization messages are passed along the tree in parallel fashion. The number of communication steps in MYSYNC mode are less than in ALLSYNC mode for small number of threads. MYSYNC mode can make progress on data transfer if some threads arrive late at the collective call. On the other hand, no progress can be made in data transfer in ALLSYNC mode. This leads to decrease in waiting time of source thread in MYSYNC mode and thus, it improves the collective time. The decrease in waiting time of other threads in MYSYNC mode lead to performance gain in overall execution time. The number of communication steps in the MYSYNC are proportional to THREADS whereas in ALLSYNC mode communication steps are proportional to $log(\texttt{THREADS})$. Thus, ALLSYNC mode performs better when compared with MYSYNC mode on large number of threads.

As per the results obtained, permute collective operation has performance improvement of almost 20% as each thread waits for only 2 threads. ALLSYNC push implementation of permute collective operation performs better than MYSYNC implementation as MYSYNC has more remote references than ALLSYNC implementation. In broadcast, scatter and gather collectives, collective time gain is considerable and overall performance improvement is close to 15%. For gatherall and exchange collectives the performance improvement is minimal as the message overhead is more as compared to message overhead in `ALLSYNC` mode.

During the course of the project we have also noticed that, as a part of future work, we can provide MuPC level non collective function `upc_notify(int)` which we can use to implement `MYSYNC` synchronization. In this function, if the argument integer is -1 then the thread which issues call to this function signals all other threads about its arrival in the collective. If the argument integer is positive then it should be less than number of threads. In that case, thread which issues call to `upc_notify(int)` signals to thread with id equal to integer passed as an argument. Using this function we can reduce the message overhead of `MYSYNC` synchronization in collectives. Thus, `MYSYNC` synchronization mode has its own space in UPC collectives.

Bibliography

[1] UPC Consortium. *UPC language specifications v1.2*, 2005.
Available at http://www.gwu.edu/ upc/docs/upc_specs_1.2.pdf.

[2] S. Chauvin, P. Saha, S. Annareddy, T. El-Ghazawi and F. Cantonnet. *UPC Manual. Technical report, George Washington university*, 2003.
Available at http://www.gwu.edu/ upc/docs/Manual-01r.pdf.

[3] E. Wiebel, D. Greenberg and S. Seidel. *UPC collective Operations Specifications v1.0. Technical report*, 2003.
Available at http://www.gwu.edu/ upc/docs/UPC_Coll_Spec_V1.0.pdf

[4] A. Mishra. *Implementing UPC collectives using GM-API. MS Project report*. Michigan Technological University, 2004.

[5] E. Fessenden. *Pairwise synchronization in UPC. MS Thesis report*. Michigan Technological University, 2004.

[6] Myrinet technical support team. *The GM message passing system. Reference guide to GM-API.*
Available at http://www.myri.com/scs/GM-2/doc/html/

[7] T. El-Ghazwi, W. Carlson, T. Sterling and K. Yelick. *UPC: Distributed shared memory programming.* Publisher : John Wiley & Sons, 2005.