Computer Science Technical Report

Pairwise synchronization in UPC

Ernie Fessenden and Steven Seidel

Michigan Technological University

Computer Science Technical Report
CS-TR-04-01
July, 2004

MichiganTech

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

Pairwise synchronization in UPC

Ernie Fessenden and Steven Seidel

July, 2004

Contents

1 Introduction **4**

2 History of UPC **4**

3 Introduction to UPC **5**

4 Introduction to the collective operations **7**
4.1 Relocalization collectives 8
4.1.1 upc_all_broadcast 8
4.1.2 upc_all_scatter 9
4.1.3 upc_all_gather 9
4.1.4 upc_all_gather_all 10
4.1.5 upc_all_exchange 10
4.1.6 upc_all_permute 11
4.2 Computational collectives 11
4.2.1 upc_all_reduce 11
4.2.2 upc_all_prefix_reduce 12
4.2.3 upc_all_sort 12

5 Synchronization modes **13**

6 Pairwise synchronization **14**
6.1 Applications of pairwise synchronization 16

7 Testbed **18**

8 Description of algorithms **19**
8.1 MYSYNC 19
8.2 Pairwise synchronization 23

9 Results **24**
9.1 Flyer 24
9.2 Lionel 28

10 Future work **34**

11 Summary **35**

Abstract

UPC (Unified Parallel C) is a small extension of 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 process that we believe to be better suited in situations where threads do not need to synchronize with all of the other threads in the system.

This project explores how to synchronize UPC threads in specialized ways. Our approach is to use UPC locks to implement pairwise synchronization. Using this new approach we implemented the MYSYNC synchronization mode and compared it to the ALLSYNC synchronization mode that is often implemented by using a barrier. Pairwise synchronization can also be used to synchronize arbitrary subsets of threads while not disturbing the remaining threads. Such subset-based synchronization does not presently exist in UPC. We also investigate how to implement the barrier using pairwise synchronization.

The MYSYNC synchronization mode has shown better performance than the ALLSYNC synchronization mode for several of the relocalization collective operations. Specifically, for an uneven computational load the MYSYNC implementation of `upc_all_broadcast`, `upc_all_scatter`, and `upc_all_gather` outperforms the ALLSYNC implementation. For `upc_all_permute` under even and uneven computational loads the MYSYNC implementation shows large improvements in run time. However, for `upc_all_exchange` and `upc_all_gather_all` the MYSYNC implementation runs slower than the ALLSYNC implementation.

1 Introduction

The goal of this thesis is to determine in what situations the MYSYNC synchronization mode is more efficient than the ALLSYNC synchronization mode. The MYSYNC synchronization mode is part of a work in progress of the UPC collectives specification [2]. This thesis also explores if pairwise synchronization is a good choice to implement the MYSYNC synchronization mode. Also, UPC does not have a simple easy to use function to facilitate a sub set barrier, this thesis explores the use of pairwise synchronization to implement one.

The implementation of the collective functions will be tested in such a way as to compare the performance of the MYSYNC synchronization mode to the ALLSYNC synchronization mode when there is an *even* workload and when there is an *uneven* workload. An even workload is defined to be when all threads have the same amount of computation to do, and an uneven workload is defined to be when one thread has twice the amount of computation as the other threads.

Section 2 covers the history of UPC. In Section 3 an overview of the UPC language is given, which includes code examples. The collective operations that we wish to apply the MYSYNC synchronization mode are reviewed in Section 4. The synchronization modes are explained in detail in Section 5. The algorithm for pairwise synchronization is in Section 6 along with an explanation of how it works. How the collective implementations were tested and timed is explained in Section 7. A description of the MYSYNC collective implementations and the sub set barrier is provided in Section 8. Section 9 covers the results that were observed for some of the collectives on Flyer and Lionel. The future work is examined in Section 10. Section 11 contains the summary. Appendix A contains the code that was added to the reference implementation of the relocalization collectives to implement the MYSYNC synchronization mode and a complete implementation of `upc_all_broadcast`. Appendix B contains all of the graphs of runtimes that were observed on Flyer.

2 History of UPC

UPC is a parallel extension to the C language that allows programmers to write parallel programs using only a small set of extensions to the C language [1]. UPC is the result of many years of work. Previous to the creation of UPC there was Split-C [3] by David Culler and Kathy Yelick and PCP [4] created by Eugene Brooks and Karen Warren. Bill Carlson created the first UPC compiler for the Cray T3E. Similar languages are Co-array Fortran [5] and Titanium [6]. These languages all have a similar memory model, in which there is a shared or global address space that all other threads or processes can access.

3 Introduction to UPC

UPC has several features that make manipulating data easy. If a thread declares a variable inside the main function such as the integer A in Figure 1, then each thread has a private variable called A in its local memory. However, UPC has a type qualifier called *shared* that signifies that the variable will reside in shared memory, such as the shared integer B in Figure 1. There is only one variable named B and it resides in the shared memory space of thread 0. All threads have access to B. However UPC can be used on a cluster where the memory is distributed and thus there is not actually any real shared memory. So whenever thread 1 or thread 2 accesses the variable B the run time system handles all of the communication necessary for each thread to get the current value of B.

For example, if thread 1 executes:

```
A = B;
```

Then the local variable A will be updated to the current value of B on thread 1.

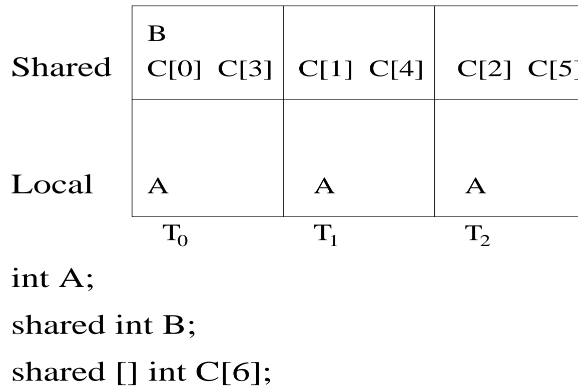


Figure 1: The UPC memory model

UPC also allows arrays to be distributed in shared memory. Immediately following the shared type qualifier the programmer may include a blocking factor for the array. The blocking factor specifies how many array elements are distributed contiguously on each thread. However, if there are more array elements than (block size * threads) the remaining elements are distributed in a cyclic fashion. For example the declaration:

```
shared [1] int C[6];
```

distributes elements 0 and 3 on thread 0, elements 1 and 4 on thread 1, and elements 2 and 5 on thread 2.

If a variable is located on a given thread then that variable is said to have *affinity* to that thread. Operations on a variable that a thread has affinity to are generally more efficient than doing operations

on variables for which the thread does not have affinity. If a shared variable is *remote*, it resides on a different thread. The reason remote operations take longer is because the run time system must do the communication necessary to get the current value of the variables involved from the thread they reside on. Operations on a shared array can be done using the upc_forall operation.

```
int i;
upc_forall(i=0; i< 6; i++; &C[i] )
    {
    C[i]++;
    }
```

The upc_forall operation works just like a for loop, except that each thread will do the body of the loop only if it has affinity to the fourth field of the upc_forall. In the example above thread 0 will increment C[0] and C[3], thread 1 will increment C[1] and C[4], and thread 2 will increment C[2] and C[5].

The programmer can have a specific thread do something different than the other threads. The identifier `MYTHREAD` is used to identify a given thread.

```
if(MYTHREAD == 0)
    {
    C[0]++;
    }
```

In this example thread 0 increments C[0].

The identifier `THREADS` can also be used, it is the number of threads in the system.

UPC has two techniques for synchronizing threads. A barrier synchronizes all threads; no thread may proceed beyond the barrier until all threads have reached the barrier.

```
upc_barrier;
```

The barrier in UPC is a *collective* operation, meaning all of the threads must call it. No thread may leave the barrier until all threads have entered the barrier. UPC also provides a split barrier, where the barrier is broken up into two commands. The first is `upc_notify` which signals all of the other threads that it has reached the synchronization point. The second is `upc_wait` which forces the threads to wait until all other threads have reached the `upc_notify`. The programmer should have useful computation following the `upc_notify` and before the `upc_wait` that uses only local data to maximize performance.

UPC provides locks to protect a specific variable.

```
#include <stdio.h>
#include <upc_strict.h>

upc_lock_t thelock;
shared int A;

main()
{
if(MYTHREAD == 0)
    {
    A = 0;
    }
upc_barrier;

upc_lock(&thelock);
A++;
upc_unlock(&thelock);

upc_barrier;
if(MYTHREAD == 0)
    {
    printf(%d \n,A);
    }
}
```

Figure 2: An Example of using locks in UPC

If the code in Figure 2 was compiled and executed thread 0 would initialize the shared variable A to 0. Then all of the threads in the system synchronize because of the statement `upc_barrier()`. After each thread gets the lock they increment the shared variable A. However as locks are mutually exclusive only one thread has the lock at a time. Then all of the threads synchronize again, and thread 0 would output the current value of A, which in this case is the value of `THREADS`.

4 Introduction to the collective operations

A collective operation is an operation that all threads must perform. There are several types of collective operations; the two types that we will focus on are the *relocalization* collectives and the *computational* collectives. The relocalization collectives copy shared data to a new location. The computational collectives do computation on data. These collectives are defined in the UPC collectives specification [2].

Figure 3 through Figure 10 use the notation T_i to signify thread i and D_i to signify data element i.

A data element can be of any arbitrary size provided it is a legal type, also a data element can be a section of an array.

4.1 Relocalization collectives

4.1.1 upc_all_broadcast

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

The `upc_all_broadcast` function copies a block of memory with affinity to a single thread to a block of shared memory on each thread. The number of bytes in each block is nbytes. If copying takes place between objects that overlap, the behavior is undefined. The `shared void *dst` is a shared pointer that is the destination of where the data is to be moved. The shared const void *src is a shared pointer that references the source of the data. The `size_t nbytes` is the number of bytes to be copied. The `upc_flag_t sync_mode` is the identifier to which synchronization mode is to be used. The synchronization mode will be explained in more detail in section 5. Figure 3 illustrates `upc_all_broadcast`.

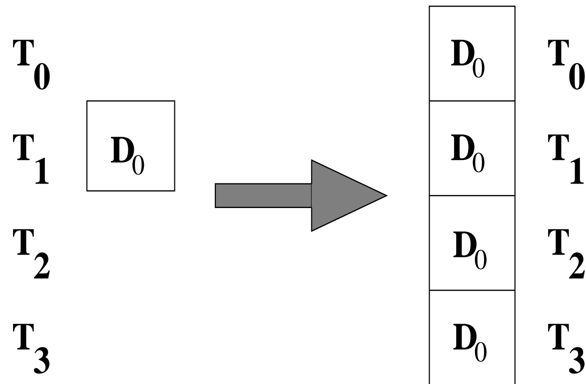


Figure 3: Data distribution of the upc_all_broadcast

D_0 already resides in shared memory, we would want to broadcast it so all threads could do local computation which should be considerably faster.

4.1.2 upc_all_scatter

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

The `upc_all_scatter` function copies the ith block of shared memory with affinity to a single thread to a block of shared memory with affinity to the ith thread. The arguments to `upc_all_scatter` and the remaining relocalization operations are similar to those for `upc_all_broadcast`. Figure 4 illustrates
`upc_all_scatter`.

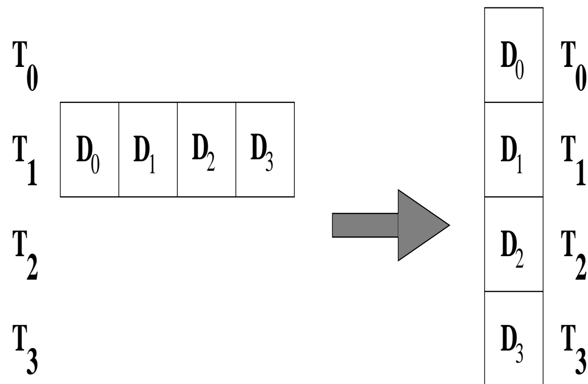


Figure 4: Data distribution of upc_all_scatter

4.1.3 upc_all_gather

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

The `upc_all_gather` function copies a block of shared memory that has affinity to the ith thread to the ith block of a shared memory area that has affinity to a single thread. Figure 5 illustrates `upc_all_gather`.

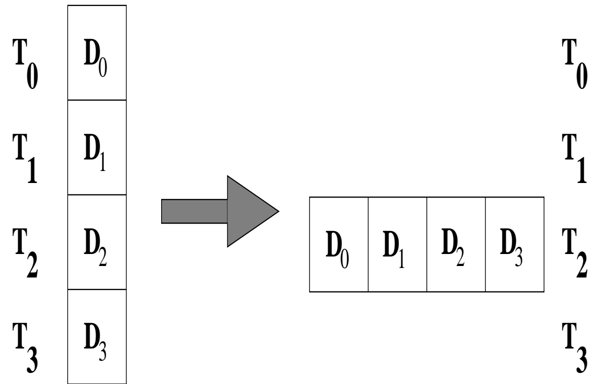


Figure 5: Data distribution of upc_all_gather

4.1.4 upc_all_gather_all

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

The `upc_all_gather_all` function copies a block of memory from one shared memory area with affinity to the ith thread to the ith block of a shared memory area on each thread. Figure 6 illustrates `upc_all_gather_all`.

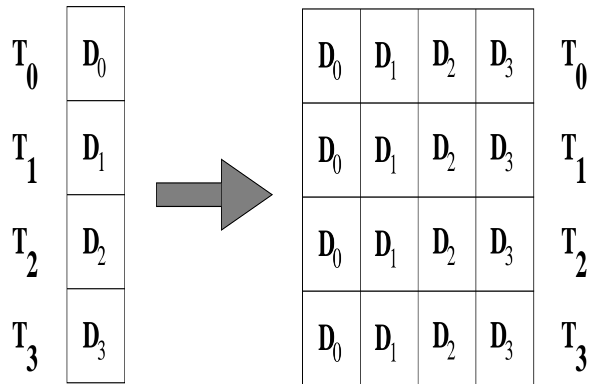


Figure 6: Data distribution of upc_all_gather_all

4.1.5 upc_all_exchange

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

The `upc_all_exchange` function copies the ith block of memory from a shared memory area that has affinity to thread j to the jth block of a shared memory area that has affinity to thread i. Figure 7 illustrates `upc_all_exchange`.

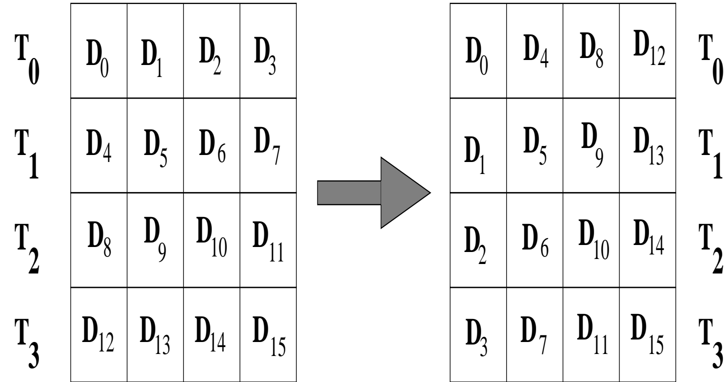


Figure 7: Data distribution of upc_all_exchange

4.1.6 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);
```

The `upc_all_permute` function copies a block of memory from a shared memory area that has affinity to the ith thread to a block of shared memory that has affinity to thread perm[i]. Figure 8 illustrates `upc_all_permute`.

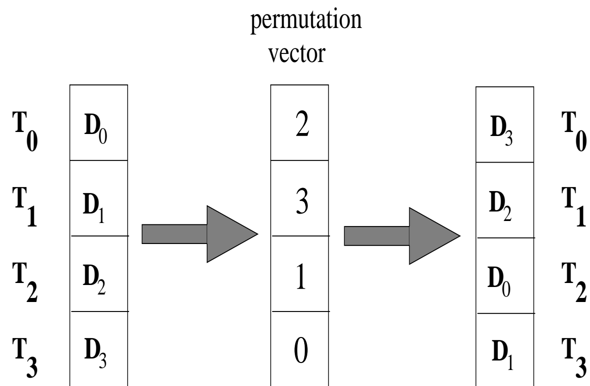


Figure 8: Data distribution of upc_all_permute

4.2 Computational collectives

4.2.1 upc_all_reduce

```
void upc_all_reduce(shared void *dst, shared const void *src,
   upc_op_t op, size_t nelems, size_t blk_size, TYPE (*func)
   (TYPE, TYPE), upc_flag_t sync_mode);
```

The `upc_all_reduce` collective operation reduces the elements in a given range of a shared array. The result of that reduction is stored in a single location. The `upc_op_t op` identifies the compu-

tational operation. For example `UPC_ADD` adds all of the elements together. The `size_t blk_size` is the number of elements in each block. Figure 9 illustrates `upc_all_reduce`.

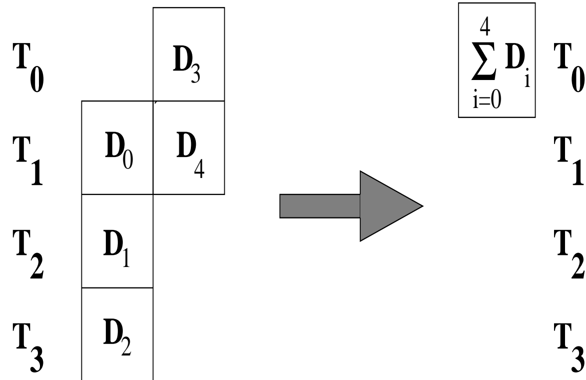


Figure 9: Data distribution of upc_all_reduce

4.2.2 upc_all_prefix_reduce

```
void upc_all_prefix_reduce(shared void *dst, shared const void *src
   upc_op_t op, size_t nelems, size_t blk_size, TYPE (*func)
   (TYPE, TYPE), upc_flag_t sync_mode);
```

The `upc_all_prefix_reduce` collective operation reduces all of the elements in a given range of a shared array, however each partial reduction is also stored. Figure 10 illustrates `upc_all_prefix_reduce`.

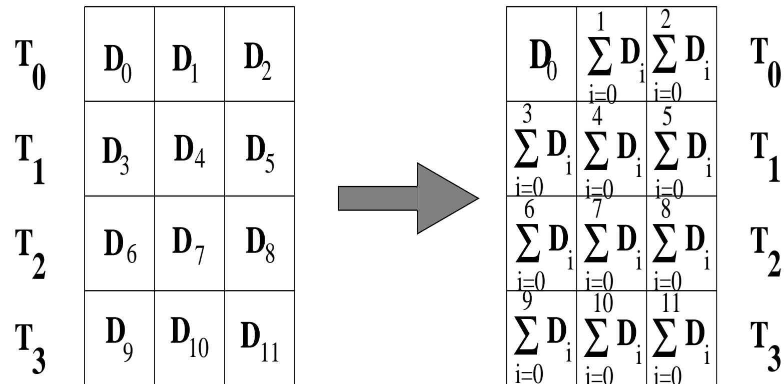


Figure 10: Data distribution of upc_all_prefix_reduce

4.2.3 upc_all_sort

```
void upc_all_sort (shared void *A, size t elem size, size_t nelems,
   size_t blk_size, int (*func)(shared void *, shared void *),
   upc_flag_t sync_mode);
```

The upc_all_sort computation collective operation sorts the elements in a given range of a shared array.

5 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 are the state of the data before and after the collective call. The following text describes the synchronization modes in detail, and the text is taken from the UPC collective operations specification V1.0 [1].

```
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, and

ALL the collective function may begin to read or write data only
    after all threads have entered the collective function call

and 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, and

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. 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, and uses a notify wait split barrier, or bundles

many exclusive collective operations together and begins and ends the sequence with a barrier.

The ALLSYNC synchronization mode is the easiest for the programmer because it provides explicit synchronization, and the programmer knows that no thread will interfere with the data being sent or received.

It is the work of this thesis 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 threads data. Our interest is in the implementation and performance of the MYSYNC synchronization mode compared to a barrier based 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 synchronization does have its place in the collective specification and will have better performance than the ALLSYNC synchronization mode in some situations.

6 Pairwise synchronization

Pairwise synchronization is the process of having two threads synchronize with each other. Ideally this operation will not have any impact on the other threads that are currently executing. This enables us to use pairs of processors synchronizing as a stepping-stone to more complex and new synchronization routines in UPC. More specifically we can use pairwise synchronization to create the MYSYNC synchronization mode. We implemented pairwise synchronization by creating a system of locks that performs a hand shaking procedure. The code that accomplishes this is seen in Figure 11.

```
upc_lock_t *prlock[THREADS][THREADS];

pairsync(int other)
    {
    upc_unlock(prlock[other][MYTHREAD])        // line 1
    upc_lock(prlock[MYTHREAD][other])          // line 2
    upc_unlock(prlock[MYTHREAD][MYTHREAD])     // line 3
    upc_lock(prlock[other][other])             // line 4
    upc_unlock(prlock[MYTHREAD][other])        // line 5
    upc_lock(prlock[other][MYTHREAD])          // line 6
    upc_unlock(prlock[other][other])           // line 7
    upc_lock(prlock[MYTHREAD][MYTHREAD])       // line 8
    }
```

Figure 11: The pairwise synchronization function

The first line is the declaration of the lock structure, which would be placed before the main function. To be able to have any thread synchronize with any other thread using pairwise synchronization a `THREADS` x `THREADS` array of locks is used, where `THREADS` is the number of threads in the system. Thus if thread 2 and thread 5 wanted to synchronize thread 2 would have to call `pairsync(5)` and thread 5 would call `pairsync(2)`.

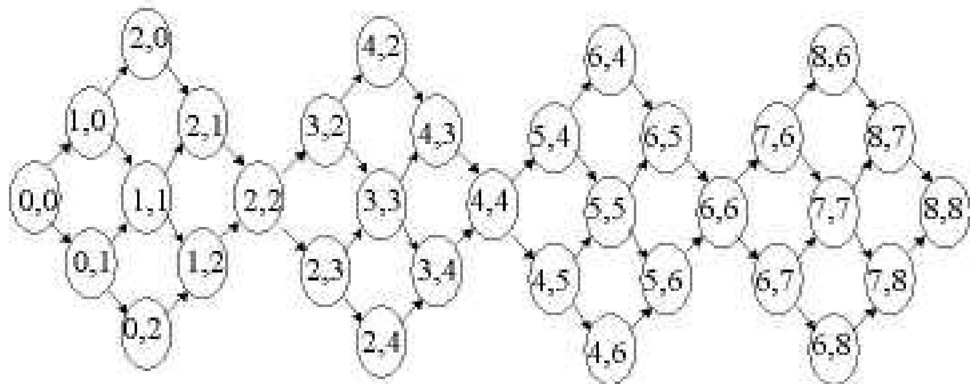


Figure 12: State diagram of the pairwise synchronization function

Because locks in UPC are mutually exclusive, two threads that are executing the pairsync function would have to navigate through the state diagram shown in Figure 12. Suppose threads 2 and 5 call the pairsync function, the numbers on the nodes represent what line of code each thread is on in the pairsync function. Initially each thread has its own column of locks in the 2d grid of locks. For example, thread 2 owns the 2nd column and thread 5 owns the 5th column of locks. When the function begins each thread has not executed any of the code in the function and thus start at the far left node of Figure 12 labeled 0,0. As thread 2 comes to its second line of code, it is waiting for lock `prlock[2][5]` to be unlocked for it to progress. However as we see thread 5 unlocks `prlock[2][5]` as its first operation, and then it waits to lock `prlock[5][2]`, which thread 2 has already unlocked. After the two threads have reached the second line of code they are synchronized and at node 2,2 of Figure 12. After both threads reach the fourth line of code they are synchronized again. The remaining code is to return the locks to their initial state so they can be re-used.

With the help of the verification tool SPIN [7] we have verified that the above code synchronizes threads. SPIN is a software tool that can be used for formal verification of software systems. This tool was developed at Bell Labs and is freely available. SPIN uses its own high-level language call PROMELA in which the programmer models their software. SPIN uses an exhaustive technique to make sure that the system is correct.

6.1 Applications of pairwise synchronization

Pairwise synchronization may be more efficient in some cases because it allows the faster threads to synchronize first, allowing them to exit the synchronization step sooner, so they can keep executing. The faster 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 be faster 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 types of 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. 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 13 illustrates these ideas. Each box represents a thread and

the arrows represent the synchronization. The top figure is for pipelines and the bottom figure is for ring structures.

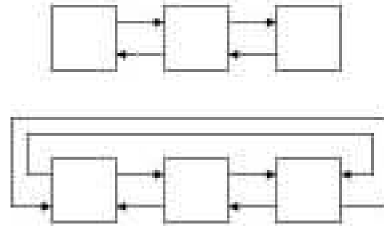


Figure 13: A possible pipeline and ring configuration synchronization pattern

Pairwise synchronization is also applicable to grids. In grid algorithms where each thread only synchronizes with its immediate neighbors there is no need for threads to synchronize threads that are not its immediate neighbor. If we used the `upc_barrier` then those threads would be synchronized. This also applies to algorithms that use a torus arrangement.

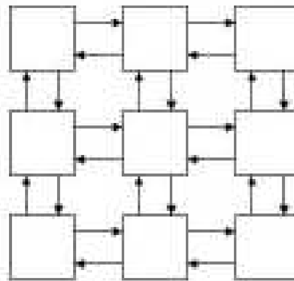


Figure 14: A possible grid configurations synchronization pattern

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 the entire system. Figure 15 illustrates a possible synchronization pattern.

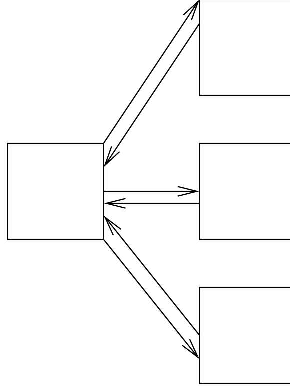


Figure 15: A possible synchronization scheme for a master drone algorithm

7 Testbed

To measure and compare pairwise synchronization techniques with existing synchronization techniques we believed that a tool with built in timing routines would be very helpful. This idea has lead us to implement a tool we call the *t*estbed where the user would first copy their code into a specific location and then compile and execute it. The user would not have to put in their own timing metrics in the code, as the testbed would already have several timing metrics already. We also wanted the testbed to be able to simulate a running environment close to how an actual run time environment would behave. Thus allowing one thread to have an uneven workload is an option, where uneven workload is defined as twice the amount of computation as the other threads. This is a general tool. It was used in this project to measure how the pairwise synchronization implementation of the MYSYNC synchronization mode compares to the ALLSYNC synchronization mode of the relocalization collectives.

The main body of the testbed is a loop where there will be a collective operation and a computation function, each timed separately. However to simulate an actual run time environment the testbed has options to skew the running threads. This means that all of the threads will not arrive at their respective collective operation or computational section of code at the same time. To do this the testbed will have parameters that control the amount of skew. Additionally the testbed has a parameter for the number of threads and for message length. The number of threads is for comparison purposes to check the performance of the collective operation as we scale up the number of threads in the system, and the message length parameter is provided so that a comparison of how the collective operations perform for various message lengths.

The testbed is a UPC application, in which the user will supply the necessary parameters to test exactly how they want their collective function to be tested. It is thought that with the testbed a user will be able to compare the effectiveness of the different synchronization modes for certain collective operations in various run time environments. In between each collective operation there is significant computation to simulate a real application. We isolate the collective communication time from the computation time and compare each run with either different synchronization modes or different implementations of the collective methods.

For purposes of this thesis the runtimes that were generated are the total time a thread was in the collective function for all the iterations. Thus the runtime for each collective call was added together for all of the iterations. Each thread recorded its own time, then only the thread that had the largest total time outputs how long it took. Therefore the results show the slowest running thread.

To better simulate a running environment and to amortize the effect of an uneven workload the thread that had the uneven workload was changed with each iteration. This was done in such a way that thread zero was never the thread with the uneven workload. Thread zero was the source for the relocalization collectives so if it had an uneven amount of work all of the other threads would have to wait for it, this situation does not perform well for MYSYNC. Also each subsequent iteration had the thread with the next higher MYTHREAD identifier be the thread which had the uneven workload, wrapping back to thread one after the thread with the highest identifier was chosen. This was done to test how any given thread with an uneven amount of computation performs instead of just a specific thread always being the thread with an uneven amount of computation.

The goal of the testbed is not only to provide a useful tool for general use, but also to find out in what situations the MYSYNC synchronization mode, which can be implemented using pairwise synchronization, is more efficient than the ALLSYNC synchronization mode.

8 Description of algorithms

8.1 MYSYNC

Originally the idea to implement the MYSYNC synchronization mode for the collective operations was to utilize a queue to keep track of what threads have entered the collective function. Then each thread would then check the queue and be able to determine what threads have entered their collective function and in what order. It is assumed that the first thread to enter into the collective function should be the first thread that should be read and or written to. In this way the first threads to enter the collective function would be able to leave before the other threads because it would be done reading and or writing all data that it has affinity to. To ensure that the threads know that all data has been read and or written each thread would synchronize with whichever thread it was reading and or writing to. Using pairwise synchronization seemed like a logical tool to use, however after several implementations were created it was determined that using pairwise synchronization was not a good general solution as it did not perform well when each thread had an even amount of computation as can be seen in Section 8.

The techniques to insure the semantics of the MYSYNC synchronization mode depend on whether the collective function is going to be a *p*ush or *p*ull based copy and thus the queue may be unnecessary. A push based implementation has each thread copy the data it has for other threads to them, while a pull based implementation has each thread copy the data it needs from other threads itself. For example, given a pull based `upc_all_broadcast` there is no need to keep track of what thread entered the collective operation first. All that is needed is for all threads other than the

source to know when the source has entered its collective operation so it knows when it can begin reading. Likewise, the source only needs to copy its own data and then check that all threads have copied from it. At this point using pairwise synchronization can increase the amount of time that other threads are waiting. If thread 1 is the source of the broadcast and there are 4 threads in the system, thread 3 could be the first to copy the data from thread 1 and then synchronize with it, at the same time threads 2 and 0 would have to wait for thread 0 to synchronize with thread 3 before they could notify thread 1 that they have also received their data. So if all the threads enter the collective function at nearly the same time, some threads will have to wait.

To to get a implementation of the MYSYNC synchronization mode that performs at least as good as the ALLSYNC implementation a strict `THREADS x THREADS` array of integers was added and the pairwise locking mechanism was entirely removed. This array is declared as strict so all read and write references to it are ordered [9]. There is a unique location within the array for each pairing of threads, and thus that particular location in the array can be checked to see if those two threads have communicated. Going back to the example of the pull based `upc_all_broadcast`, each thread needs to write to its location on the source's row of the array after it copies the data and it can then exit the collective function. The source needs to go through its row of the array and wait until all threads have marked that they are done and then it can exit the collective function. The `upc_all_gather` and `upc_all_scatter` are similar in implementation.

This method requires that the writes to the array be written back to memory and not just cached. If the writes were only cached then the other threads would have to wait longer than necessary. All run time systems that were used during this project did write backs of strict operations, however the UPC specification does not require it [9].

The required synchronization to facilitate MYSYNC for `upc_all_exchange` is slightly different. The push and pull based implementation are the same, except for the actual parameters to `upc_memcpy()`. However the queue is still used to keep track of what order the threads entered the collective function. What is interesting, however, is the exit synchronization. Because all threads need to read and write to all other threads no thread can leave until all other threads have entered the collective function and have begun reading and writing. If there are 4 threads in the system and thread 2 is late entering the collective, then threads 0, 1, and 3 have already finished reading and writing to each other. Assuming a pull based implementation, once thread 2 finally enters the collective all threads are able to read from it, however no thread is able to leave until thread 2 reads from them. Thus the semantics of `UPC_OUT_MYSYNC` are not substantially different than those of `UPC_OUT_ALLSYNC`. Similar observations apply to `upc_all_gather_all`.

On the other hand `upc_all_permute` is able to take very good advantage of the MYSYNC semantics. Given that each thread only needs to know if the thread that they need to read or write to has entered the collective operation, a single late thread will adversely effect at most two other threads. Once a thread knows that the thread it needs to read or write to has entered the collective, it can proceed to read or write, and then it writes in the array notifying that thread that their data has been received or sent. Then each thread needs to wait until it has been notified that it has received its data or it has had its data read.
Table 1 shows the total number of *communication steps* necessary to facilitate the respective

Collective	Total Messages	
	ALLSYNC	MYSYNC
upc_all_broadcast	THREADS-1	THREADS-1
upc_all_scatter	THREADS-1	THREADS-1
upc_all_gather	THREADS-1	THREADS-1
upc_all_exchange	THREADS-1	THREADS*(THREADS-1)
upc_all_gather_all	THREADS-1	THREADS*(THREADS-1)
upc_all_permute	THREADS-1	THREADS

Table 1: Communication steps to facilitate synchronization

synchronization mode. A communication step is defined as a request and a reply message. In terms of the number of communication steps, the MYSYNC implementation is comparable for `upc_all_broadcast`, `upc_all_scatter`, `upc_all_gather`, and `upc_all_permute`. However `upc_all_exchange` and `upc_all_gather_all` have a factor of `THREADS` more synchronization steps.

Collective	Maximum parallelism	
	ALLSYNC	MYSYNC
`upc_all_broadcast`	2*⌈lg(THREADS)⌉	THREADS-1
`upc_all_scatter`	2*⌈lg(THREADS)⌉	THREADS-1
`upc_all_gather`	2*⌈lg(THREADS)⌉	THREADS-1
`upc_all_exchange`	2*⌈lg(THREADS)⌉	THREADS-1
`upc_all_gather_all`	2*⌈lg(THREADS)⌉	THREADS-1
`upc_all_permute`	2*⌈lg(THREADS)⌉	2

Table 2: Minimum number of communication steps in parallel

Table 2 shows the number of communication steps required if all of the communication can be done with maximum parallelism. It is assumed that the ALLSYNC implementation is based upon a binary tree barrier. It is interesting that the MYSYNC implementation can simultaneously be sending data and synchronizing with other threads. if `upc_all_barrier` is used to implement the ALLSYNC mode, then data must be sent separately. In this way the MYSYNC implementation can see better performance for the collectives because some of the synchronization messages can be sent at the same time as data. Also, in the case of `upc_all_permute` the number of synchronization steps that can be done in parallel is constant.

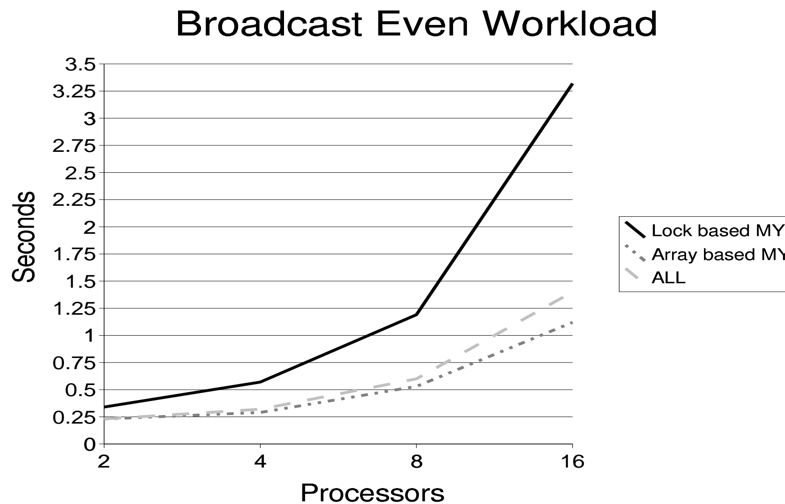


Figure 16: Flyer run times of a pull-based `upc_all_broadcast` for 1000 iterations.

As can be seen in Figure 16 the performance of the lock-based implementation of the MYSYNC synchronization mode is worse than if all of the threads used a

barrier that would satisfy the ALLSYNC requirements. However the array-based implementation of MYSYNC does improve performance compared to ALLSYNC.

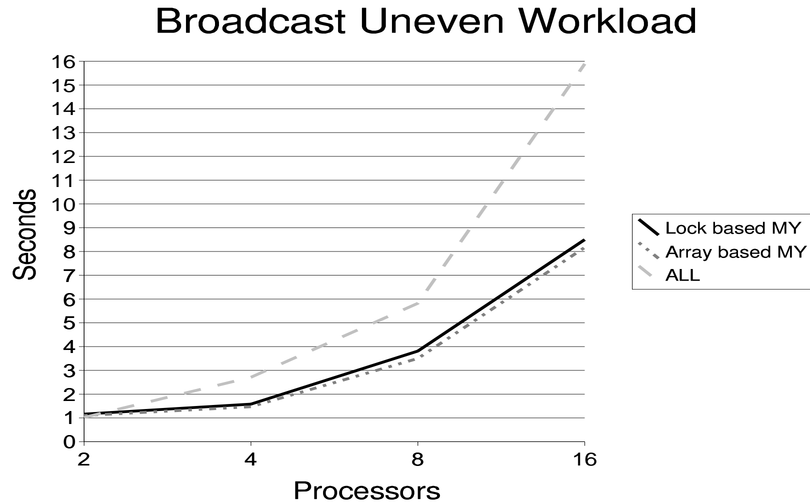


Figure 17: Flyer run times of a pull-based `upc_all_broadcast` for 1000 iterations.

When one thread has twice as much work as the other threads in the system the lock-based implementation of MYSYNC performs better than a barrier implementation of ALLSYNC as can be seen in Figure 17. It should also be noted that the array-based implementation outperforms the lock-based implementation. A more detailed overview of the results is presented in Section 9.

8.2 Pairwise synchronization

Although pairwise synchronization was not found to be a good general solution for MYSYNC it is still useful for other algorithms. As covered in Section 6.1, pairwise synchronization can be used to implement less stringent synchronization schemes than using the `upc_barrier`. However it is also interesting that a lock-based barrier can be implemented using pairwise synchronization.
This algorithm has all threads synchronize with thread 0 using pairwise synchronization, and then synchronize again. Because thread 0 synchronizes with all threads in order, once a thread is able to synchronize with thread 0 the second time it knows that all other threads have reached the `upc_all_pairwise_barrier` operation.

Similarly, a *subset barrier* can be created using pairwise synchronization. A subset barrier is one in which only a specified subset of threads are synchronized. One way to implement this is to have all threads send an array that contains all of the threads in the subset. However to ensure that the threads will not dead lock in their pairwise synchronization calls the array has to be sorted. Then, instead of going through a loop for all threads like the global barrier, each thread would loop

```
void upc_all_pairwise_barrier()
{
int i;

// thread 0 needs to synchronize with all other threads
if (MYTHREAD == 0)
    {
    for(i=1;i<THREADS;i++)
        {
        pairlock(i);
        }
    for(i=1;i<THREADS;i++)
        {
        pairlock(i);
    }
}
// all threads synchronize with thread 0
else
    {
    pairlock(0);
    pairlock(0);
    }
}
```

Figure 18: A barrier implemented using pairwise synchronization

through the array that was passed and synchronize with those threads, excluding itself.

9 Results

9.1 Flyer

The majority of the work that was done for this project was done on Flyer. Flyer is a thirty-two processor Compaq SC40 which consists of eight four processor Alpha SMP's connected by a Quadrics switch. Compaq's UPC compiler, Field Test version 2.2 was used to compile the programs. The test cases only tested for two, four, eight, and sixteen processors. Although Flyer has thirty-two processors scaling the tests up to thirty-two would not have yielded good results because then all four processors on each node would be used and the system processes would have interrupted the testbed [8].

As was seen in section 8.1 the lock-based implementation of pairwise synchronization did not perform well in the case when all the threads in the system have the same amount of computation to

do between each collective call. The strict array implementation of the MYSYNC synchronization mode was explored to see if its performance was better. The goal was to obtain approximately the same performance as the ALLSYNC synchronization mode when there is approximately the same amount of work, and to have improved performance when there is one slow thread.

The first collective operation to be implemented was `upc_all_broadcast`. It was used as a basis of study until it was determined that the implementation of the MYSYNC synchronization mode using a strict array was capable of delivering improved performance.

All tests were done where each thread executes its collective operation and then performs some artificial computation. When there is an *uneven* workload, one thread does its artificial computation twice. This procedure was done in a loop for 1,000 iterations and the reported times are for the slowest thread. Each thread keeps track of its total collective time by summing the time it has spent in the collective each iteration. 1,000 iterations was chosen to insure the times were much larger than the granularity of the system clock.

Using the MYSYNC synchronization mode for `upc_all_broadcast` provides better performance than a barrier implementation of the ALLSYNC synchronization mode as can be seen in Figure 19. Provided that the source thread is not the slow thread, the MYSYNC synchronization mode enables threads to read from the source as soon as they enter the collective function. Once a thread has read from the source it needs to notify the source that it has received its data and can then exit the collective. An ALLSYNC synchronization implementation requires that all threads need to enter the collective operation and then reading and writing can begin. Since the MYSYNC synchronization mode is more relaxed when there is a slow thread, only the slow thread and the source are affected. Under an even workload the strict array implementation of the MYSYNC synchronization mode performs approximately as well as the barrier implementation of the ALLSYNC synchronization mode. Similar results can be seen for `upc_all_gather` and `upc_all_scatter`. See Appendix A for complete run times.

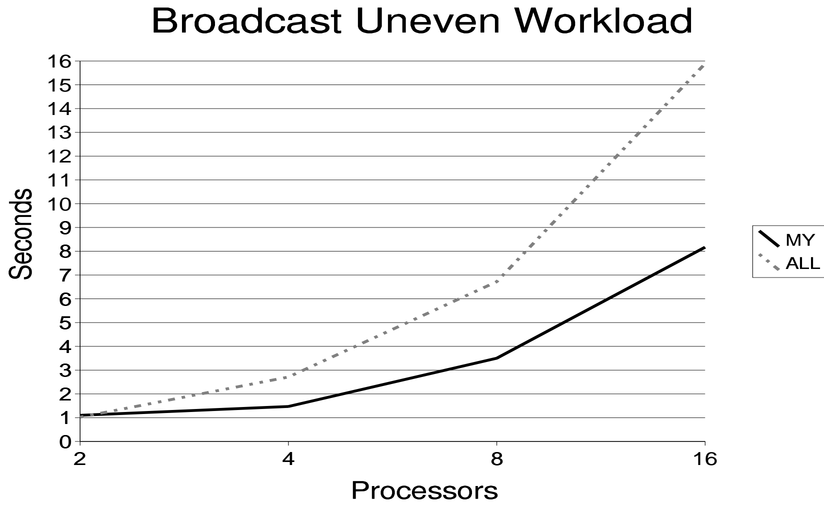


Figure 19: Flyer run times of pull-based `upc_all_broadcast` for 1000 iterations

The results for a pull-based implementation of `upc_all_exchange` show that when each thread needs to communicate with all other threads in the system the performance of the strict array implementation of the MYSYNC synchronization mode does not improve performance. This is because if there is a slow thread in the system all threads need to wait for the slow thread so they can copy their data from it. Figure 20 shows the run times of `upc_all_exchange`. Similar results for `upc_all_gather_all` are given in Appendix A.

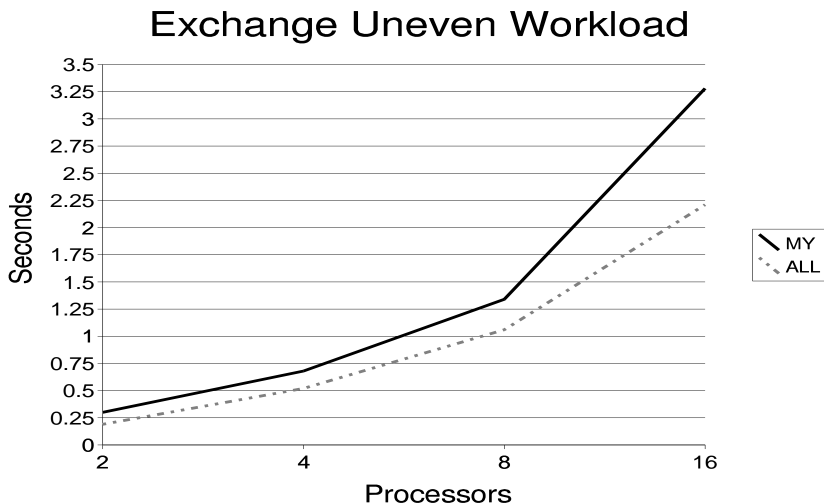


Figure 20: Flyer run times of pull-based `upc_all_exchange` for 1000 iterations

A significant performance increase was achieved in the strict array implementation of the MYSYNC synchronization mode for `upc_all_permute`. This is because `upc_all_permute` requires much less communication among all the threads. Each thread has at most two other threads it needs to communicate with, the thread it needs to send its data to and the thread it needs to receive data from. So if there is one slow thread in the system it will affect at most two other threads besides itself. In addition, even when there is a even workload the MYSYNC implementation of `upc_all_permute` outperforms the ALLSYNC implementation. Figure 21 shows the performance of `upc_all_permute`.

Push-based implementations of all of relocalization collective operations were implemented. They show performance similar to the pull-based implementations. This data is given in Appendix A.

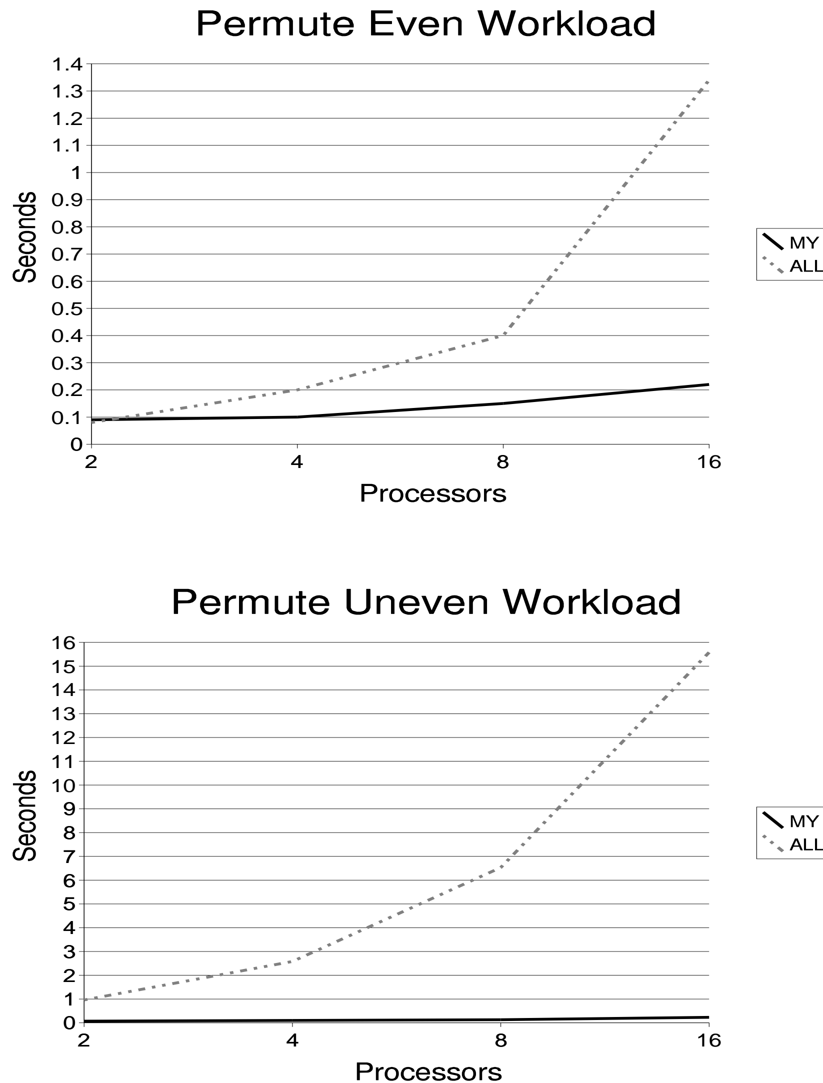


Figure 21: Flyer run times of pull-based `upc_all_permute` for 1000 iterations

9.2 Lionel

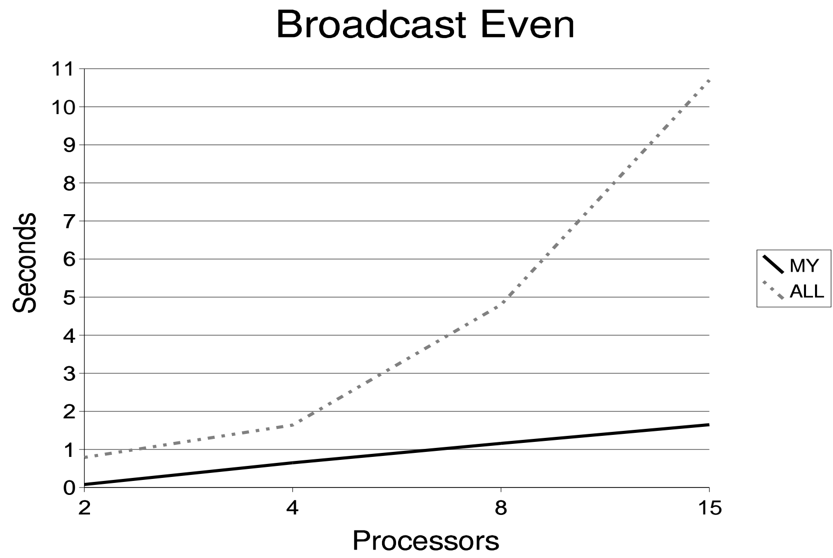
Although the majority of the testing was done on Flyer, Lionel was also used for testing purposes. Lionel is a 16 node Beowulf cluster connected by a fiber optic Myrinet switch. Each node is a dual processor 2GHz Pentium 4 with 2GB of memory. The MuPC compiler version number 1.0.7pre and the GASNet compiler version number 1.1.1 were used. To conserve time on Lionel the number of iterations was reduced. Also, at the time one node was unavailable so the implementations were only tested up to fifteen processors.

The run times for MTU's MuPC compiler were slightly different than the ones observed on Flyer. Since MuPC treats local shared pointers the same as it does for remote shared pointers, the overall times were much slower. The MYSYNC implementation of `upc_all_broadcast` showed good performance compared to the ALLSYNC version for both the even workload and uneven workload. Also, the uneven workload did not skew the run times as much as it did on Flyer.

The results for `upc_all_exchange` were similar to that of Flyer. The MYSYNC implementation was slower than the ALLSYNC implementation. However the uneven workload skewed the run times on `upc_all_exchange` on Lionel much more than it did on Flyer.

For both an even and uneven workload the MYSYNC implementation of `upc_all_permute` performed much better than that of the ALLSYNC implementation. Also, the uneven workload did not skew the run times as much on Lionel as it did on Flyer, which can be seen in Figure 24.

The following are run times using MTU's MuPC compiler.



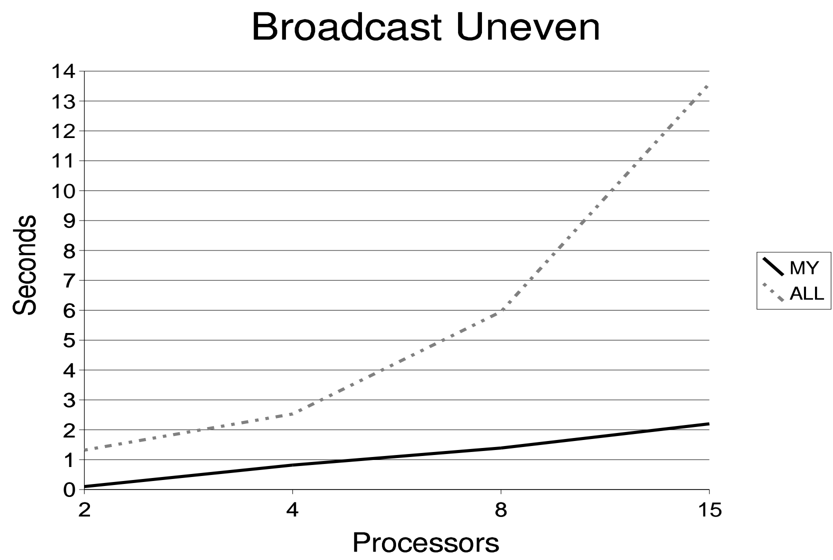
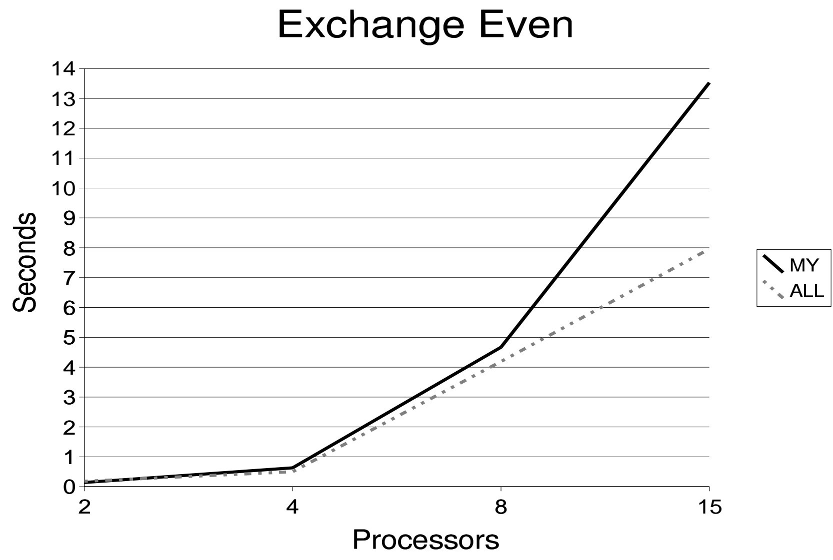


Figure 22: MuPC run times of pull-based `upc_all_broadcast` for 200 iterations.



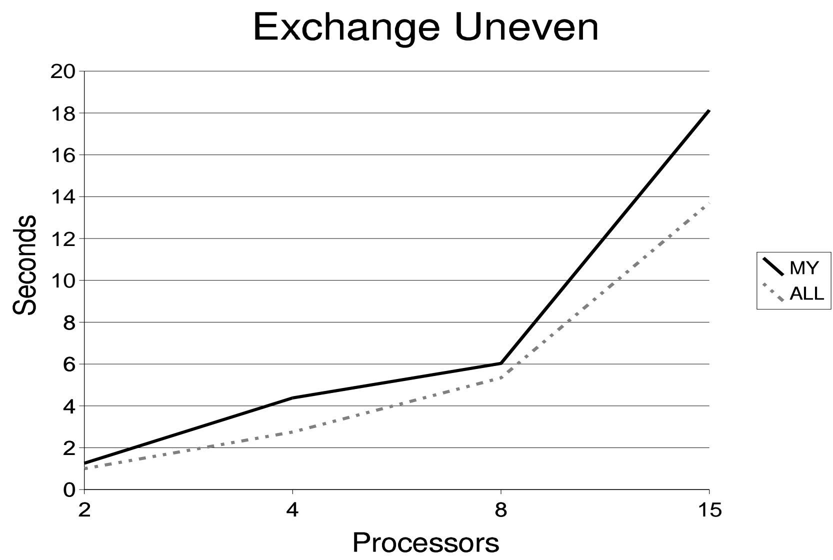
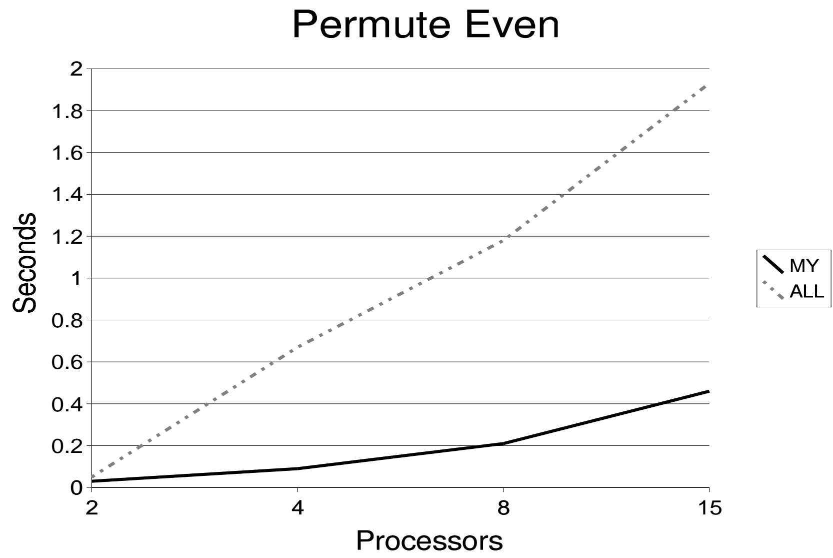


Figure 23: MuPC run times of pull-based `upc_all_exchange` for 200 iterations.



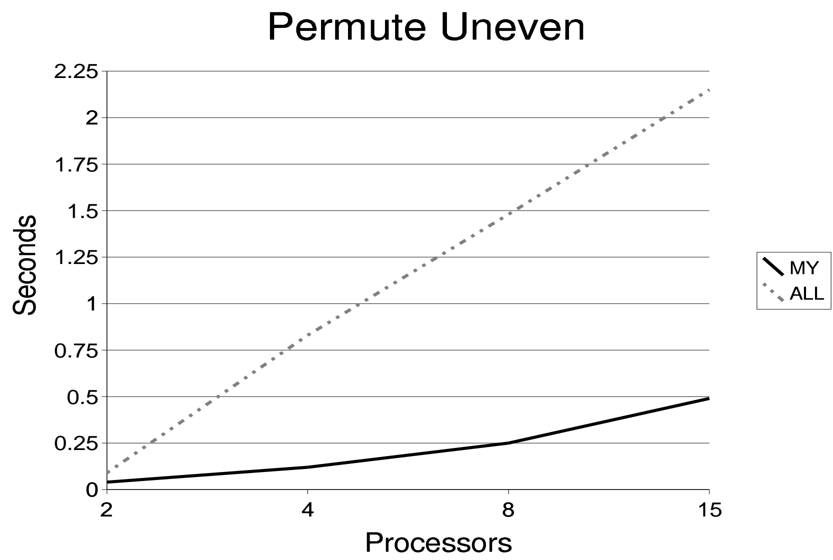
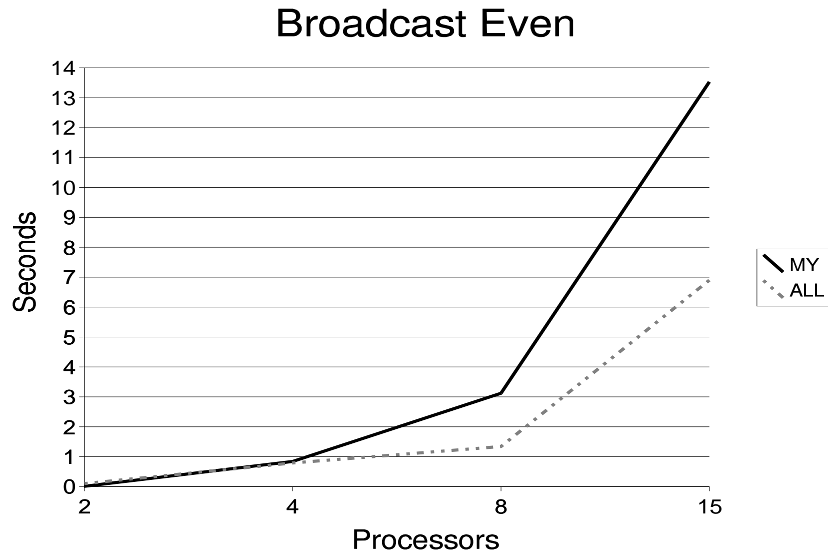


Figure 24: MuPC run times of pull-based `upc_all_permute` for 200 iterations.

The following are run times using Berkeley's GASNet compiler.



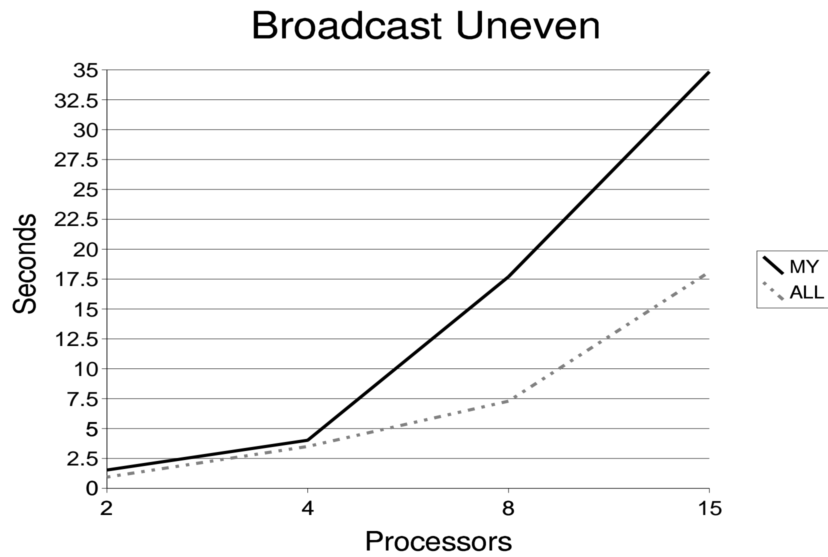
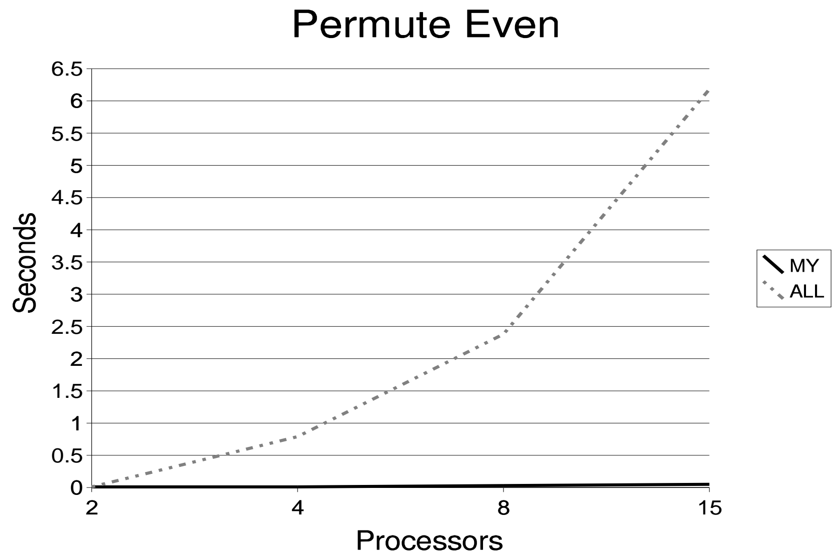


Figure 25: GASNet run times of pull-based `upc_all_broadcast` for 20 iterations.



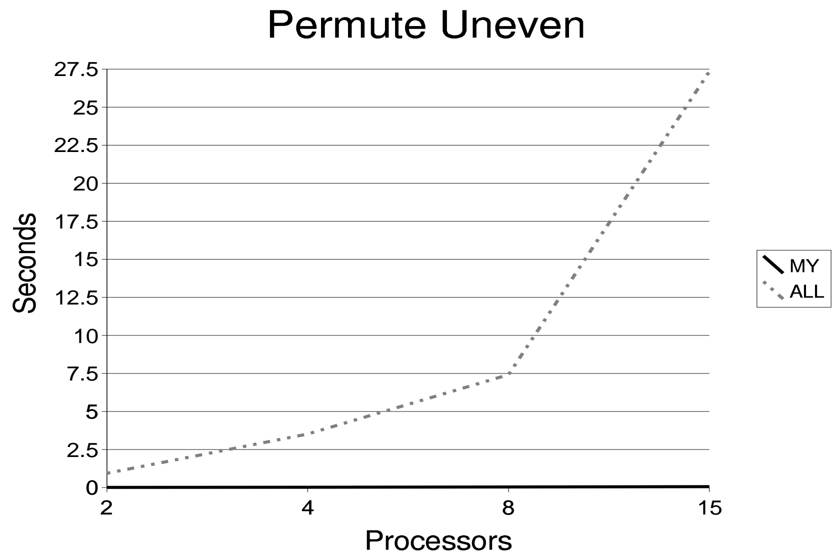


Figure 26: GASNet run times of pull-based `upc_all_permute` for 20 iterations.

The run times for Berkley's compiler on the cluster were much slower than that of HP's compiler on Flyer. However for `upc_all_broadcast` the trend of the MYSYNC implementation outperforming the ALLSYNC implementation continued. Also, similar to performance on Flyer, the uneven workload skewed the run times significantly, as can be seen in Figure 25.

The performance of `upc_all_permute` using Berkley's compiler on the cluster showed results similar to Flyer as can be seen in Figure 26. The uneven workload skewed the run times for the ALLSYNC implementation but did not effect the MYSYNC implementation nearly as much.

10 Future work

During the course of this work several questions were answered and several ideas about new work arose. There are several other techniques that could have been used to implement the MYSYNC synchronization mode for the collective operations that were not investigated. There may also be techniques to improve the performance and functionality of pairwise synchronization. Some of those questions and techniques are:

1. **Explore if a queue is necessary**. It was assumed that the queue was needed to facilitate which order the threads entered the collective operation so that threads would be able to read and/or write to the threads in the order they arrived so that the first threads could leave sooner. However it may be possible that another scheme is more efficient.

2. **Provided that the queue is necessary, determine another mechanism to facilitate it**. Currently the queue uses a lock to ensure that only one thread at a time is able to update the queue. This system does not scale well and a better solution may be possible.

3. **Maintaining a local bit vector of which threads have been read and or written to**. This may be more efficient. The existing system uses polling to read the remote values of the strict array which can lead to poor performance [8].

4. **Implement a back off scheme**. The current implementation uses polling to determine if other threads have read and or written their data. It may be possible to implement a back off scheme in which a thread that detects that another thread is not ready goes into a wait loop so it does not relentlessly poll the other thread and thereby degrade performance [8].

5. **Compare combinations of synchronization modes**. A complete comparison of all nine combinations of synchronization modes would be useful for future programmers to determine which synchronization mode they wish to use for specific situations.

6. **Explore platform specific implementations**. The collective operations were written at the UPC level. A lower level implementation, possibly even platform specific implementations, would most likely yield improved performance.

7. **Explore tree based implementation of the MYSYNC synchronization mode**. It may be possible to decrease the number of communication steps that can be done in parallel by converting the existing algorithm to use a tree structure.

8. **Explore tree based implementation of sub set barrier**. The implementation of the pairwise barrier was done in a linear fashion. A tree implementation should prove to have much better performance. A comparison between the existing `upc_barrier` and the tree pairwise barrier would be of interest.

11 Summary

Using pairwise synchronization through a handshaking procedure of locks opens the door to new algorithm designs in UPC. Although pairwise synchronization could be used to implement the MYSYNC synchronization mode of the collective operations it was found that a strict array implementation was more efficient. In the case of `upc_all_permute` the MYSYNC synchronization mode has proven to be much more efficient than the ALLSYNC synchronization mode. Similarly when one thread has a larger workload than the other threads MYSYNC is efficient than ALLSYNC for `upc_all_broadcast`, `upc_all_gather`, and `upc_all_scatter`. However, the results for `upc_all_exchange` and `upc_all_gather_all` show that the `UPC_OUT_MYSYNC` portion of the synchronization mode is similar to the `UPC_OUT_ALLSYNC` synchronization mode when all threads need to read and write to all other threads and therefore it does not improve performance when there is a slow thread.

Appendix A

Code base

This appendix includes the items that would need to be added to the reference implementation's `upc_collective.c` file, and a complete implementation of the `upc_all_broadcast`.

```
#include <upc.h>
#include "upc_collective.h"

// The following was added to the original upc_collective.c

// struct to hold thread number and collective count
struct upc_first
{
int count;
int thread;
};

// queue to keep track of what thread arrives first in their collective call
shared [1] struct upc_first upc_order[THREADS];

// array for notification that data has been recieved
shared [1] strict int strict_upc_ready[THREADS][THREADS];

// array to keep track of collective count
shared [1] strict int strict_upc_count[THREADS];

int upc_coll_init_flag = 0;

// lock for the queue
upc_lock_t *readylock;
```

```
void upc_all_broadcast(shared void *dst,shared const void *src,size_t
    nbytes, upc_flag_t sync_mode )
{
int i;
int initiator;
int local_count;
struct upc_first local_first;

if ( !upc_coll_init_flag ) {
    upc_coll_init();
    }

#ifdef _UPC_COLL_CHECK_ARGS
upc_coll_err(dst, src, NULL, nbytes, sync_mode, 0, 0, 0,
            UPC_BRDCST);
#endif

#ifndef PULL
#ifndef PUSH
#define PULL TRUE
#endif
#endif

// increment the count of collectives
strict_upc_count[MYTHREAD]++;
local_count = strict_upc_count[MYTHREAD];

initiator = upc_threadof((shared char *) src);

// Synchronize using barriers in the case ALLSYNC.
if ( ~(UPC_IN_NOSYNC & sync_mode) ) {
    upc_barrier;
    }

#ifdef PULL

// Each thread "pulls" the data from the source thread.
if (UPC_IN_MYSYNC & sync_mode) {
    while(1) {
        if (local_count == strict_upc_count[initiator]) {
            upc_memcpy((shared char *)dst + MYTHREAD,(shared char *)
                src,nbytes);
            break;
            }
        }
```

```
    }
else {
    upc_memcpy((shared char *)dst + MYTHREAD,(shared char *)src,
        nbytes);
    }

// wait for all threads to receive the data in the case of MYSYNC
if (UPC_OUT_MYSYNC & sync_mode) {
    if (MYTHREAD == initiator) {
        strict_upc_ready[initiator][MYTHREAD] = 1;
        for(i = 0; i <THREADS;i++) {
            if (strict_upc_ready[initiator][i] == 1) {
                strict_upc_ready[initiator][i] = 0;
                }
            else {
                i--;
                }
            }

        }
    else {
        strict_upc_ready[initiator][MYTHREAD] = 1;
        }
    }

#endif

#ifdef PUSH

// The source thread "pushes" the data to each destination.
// Enter thread number into the queue in case of MYSYNC
if (UPC_IN_MYSYNC & sync_mode) {
    upc_lock(readylock);
    for(i = 0; i < THREADS; i++) {
        local_first = upc_order[i];
        upc_fence;
        if (local_first.count < local_count) {
            upc_order[i].thread = MYTHREAD;
            upc_order[i].count = local_count;
            upc_fence;
            break;
            }
        }
    upc_unlock(readylock);
```

```
    if ( initiator == MYTHREAD ) {
        for (i=0; i<THREADS; i++) {
            local_first = upc_order[i];
            if(local_first.count == local_count) {
                upc_memcpy((shared char *)dst + local_first.thread,
                    (shared char *)src,nbytes);
                // Notify thread that the data has been received
                if (UPC_OUT_MYSYNC & sync_mode) {
                    strict_upc_ready[initiator][local_first.thread] = 1;
                    }
                }
            else {
                i--;
                }
            }
        }
    }
else {
    if ( initiator == MYTHREAD ) {
for (i=0; i<THREADS; ++i) {
    upc_memcpy((shared char *)dst + i,(shared char *)src,nbytes);
            if (UPC_OUT_MYSYNC & sync_mode) {
                strict_upc_ready[initiator][i] = 1;
                }
    }
        }
    }
// wait for all threads to receive the data in the case of MYSYNC
if (UPC_OUT_MYSYNC & sync_mode) {
    while(1) {
        if (strict_upc_ready[initiator][MYTHREAD] == 1) {
            strict_upc_ready[initiator][MYTHREAD] = 0;
            break;
            }
        }
    }
#endif

// Synchronize using barriers in the case of ALLSYNC.
if ( ~(UPC_OUT_NOSYNC & sync_mode) ) {
    upc_barrier;
    }
}
```

Appendix B

Run times

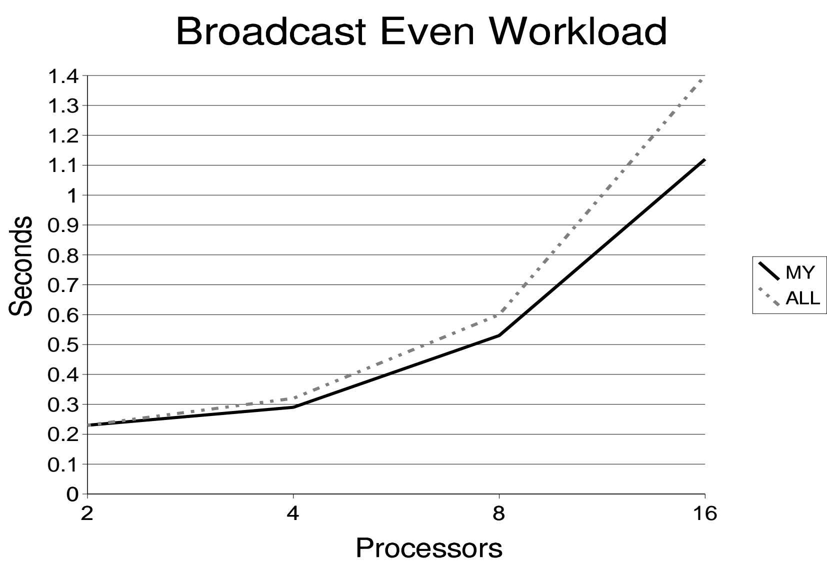
The following run times were generated using Field Test Version 2.2 of Compaq's compiler on Flyer. Additionally a comparison of the number of communication steps in the respective collective operations is presented.

Collective	Total Messages	
	ALLSYNC	MYSYNC
upc_all_broadcast	THREADS-1	THREADS-1
upc_all_scatter	THREADS-1	THREADS-1
upc_all_gather	THREADS-1	THREADS-1
upc_all_exchange	THREADS-1	THREADS*(THREADS-1)
upc_all_gather_all	THREADS-1	THREADS*(THREADS-1)
upc_all_permute	THREADS-1	THREADS

Table 3: Communication steps to facilitate synchronization

Collective	Maximum parallelism	
	ALLSYNC	MYSYNC
upc_all_broadcast	$2*\lceil \lg(\texttt{THREADS}) \rceil$	THREADS-1
upc_all_scatter	$2*\lceil \lg(\texttt{THREADS}) \rceil$	THREADS-1
upc_all_gather	$2*\lceil \lg(\texttt{THREADS}) \rceil$	THREADS-1
upc_all_exchange	$2*\lceil \lg(\texttt{THREADS}) \rceil$	THREADS-1
upc_all_gather_all	$2*\lceil \lg(\texttt{THREADS}) \rceil$	THREADS-1
upc_all_permute	$2*\lceil \lg(\texttt{THREADS}) \rceil$	2

Table 4: Minimum number of communication steps in parallel



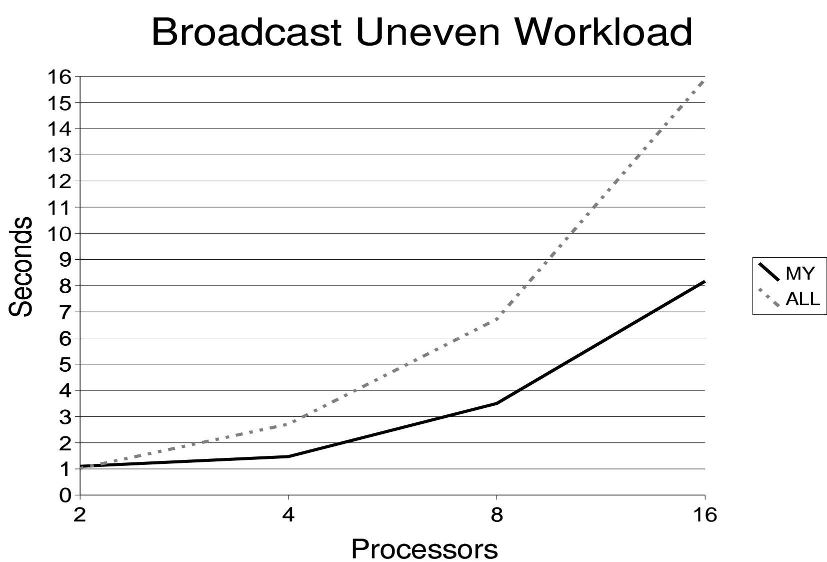


Figure 27: Flyer run times of pull-based upc_all_broadcast for 1000 iterations.

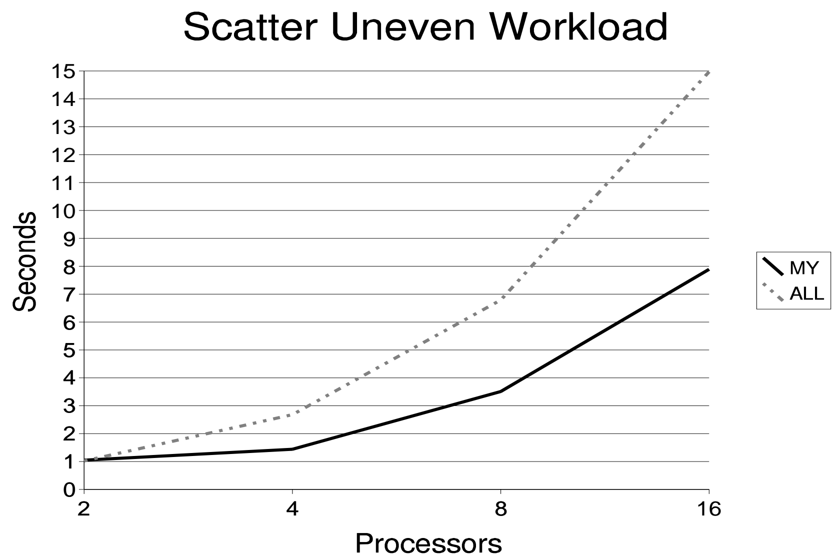
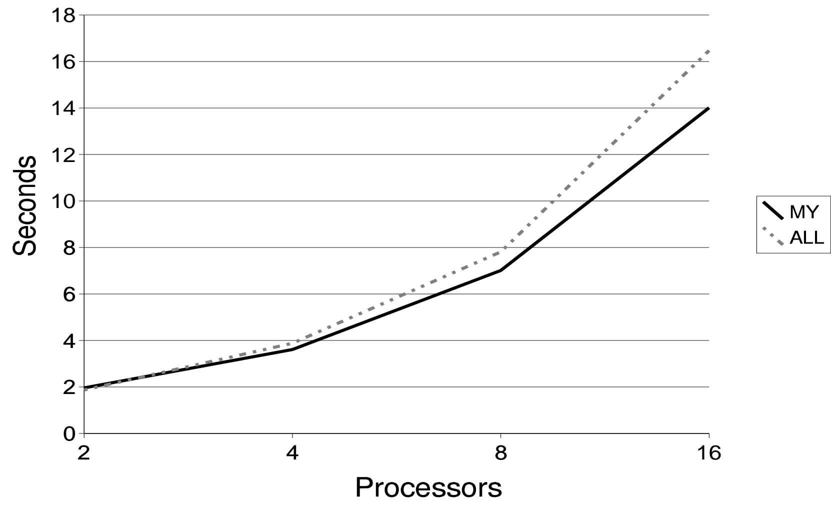


Figure 28: Flyer run times of a pull-based upc_all_scatter for 1000 iterations.



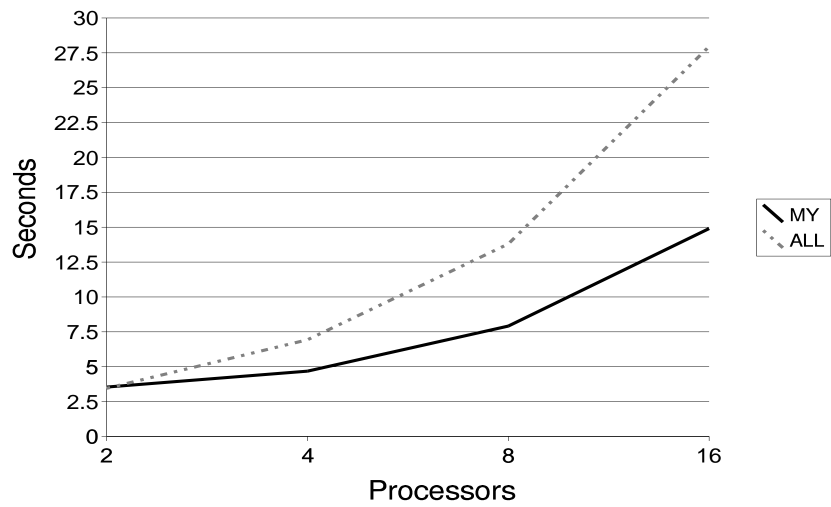


Figure 29: Flyer run times of pull-based upc_all_gather 1000 iterations.

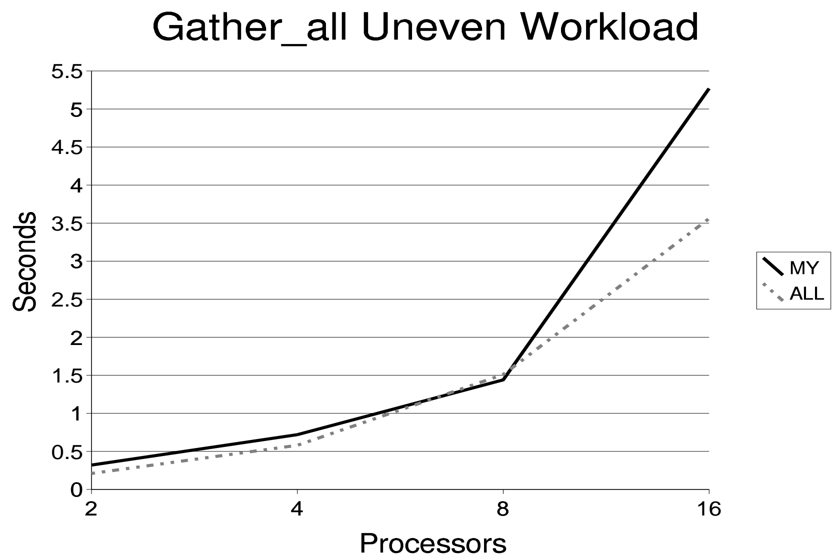


Figure 30: Flyer run times of pull-based upc_all_gather_all for 1000 iterations.

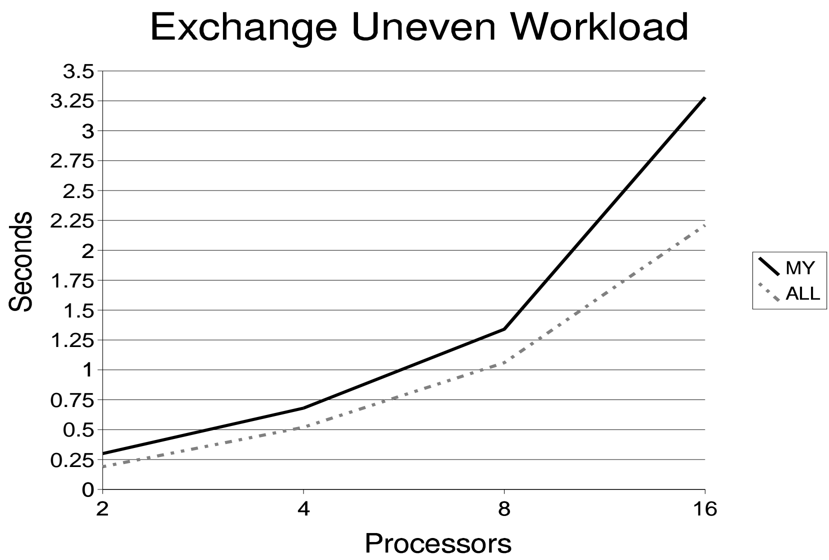
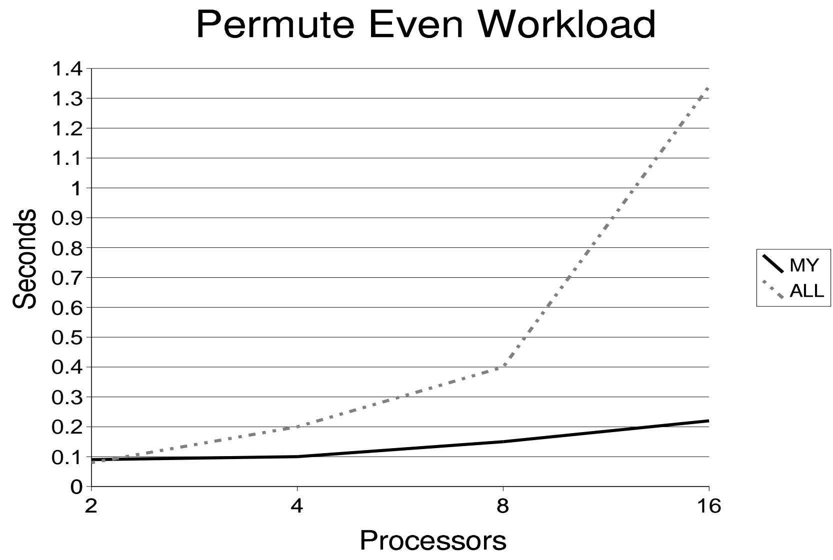


Figure 31: Flyer run times of a pull-based upc_all_exchange for 1000 iterations.



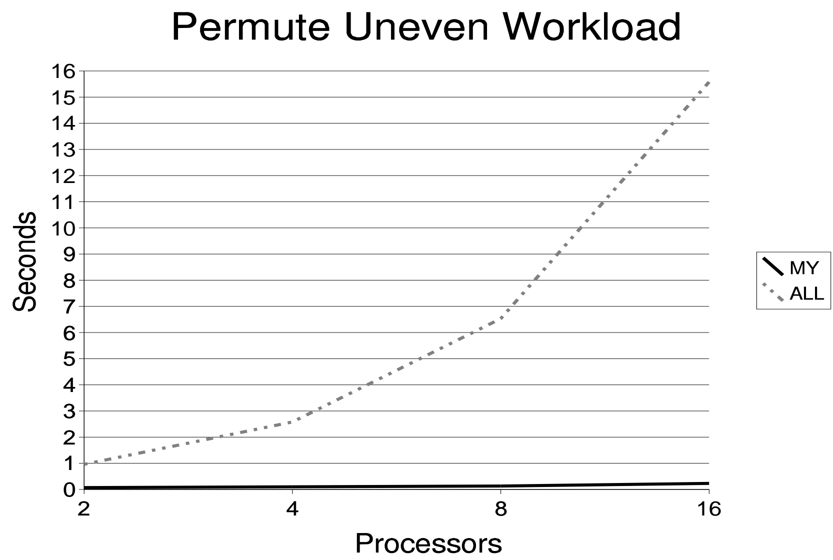
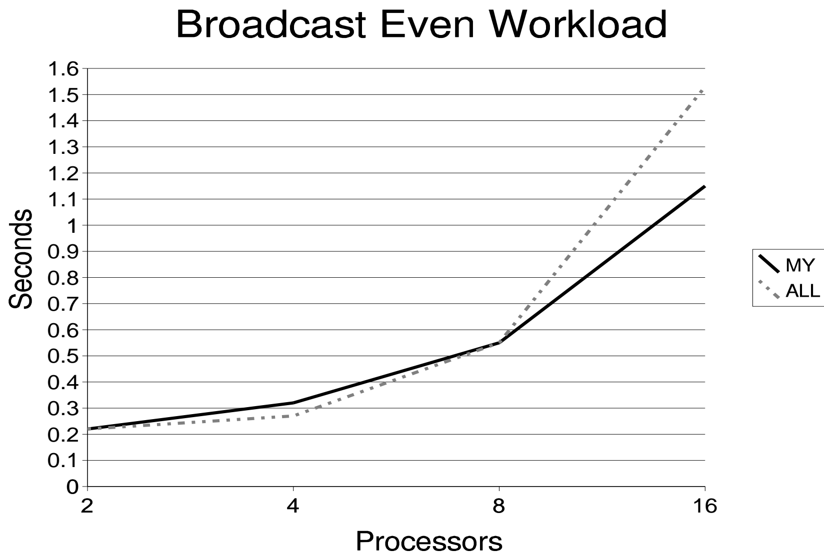


Figure 32: Flyer run times of a pull-based upc_all_permute for 1000 iterations.



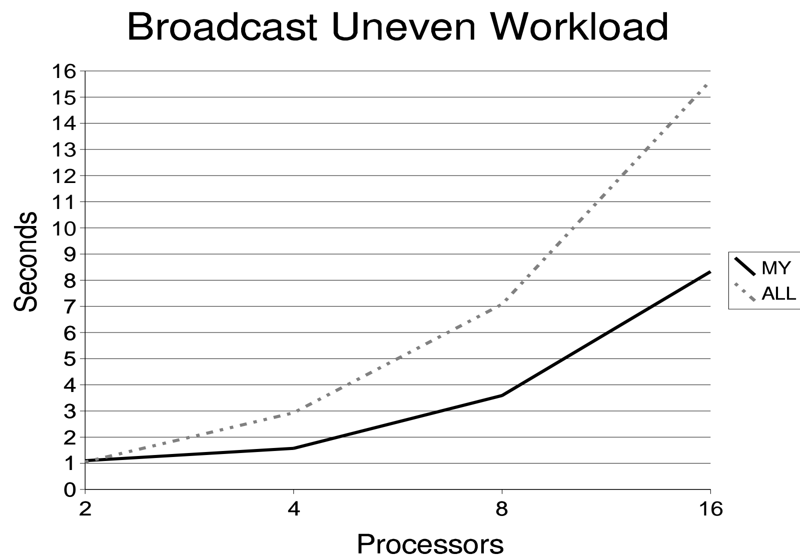


Figure 33: Flyer run times of push-based upc_all_broadcast for 1000 iterations.

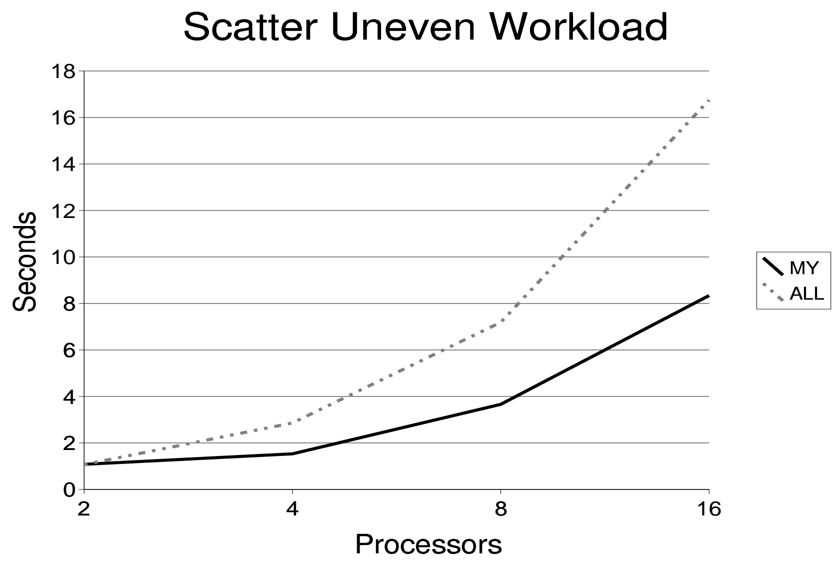
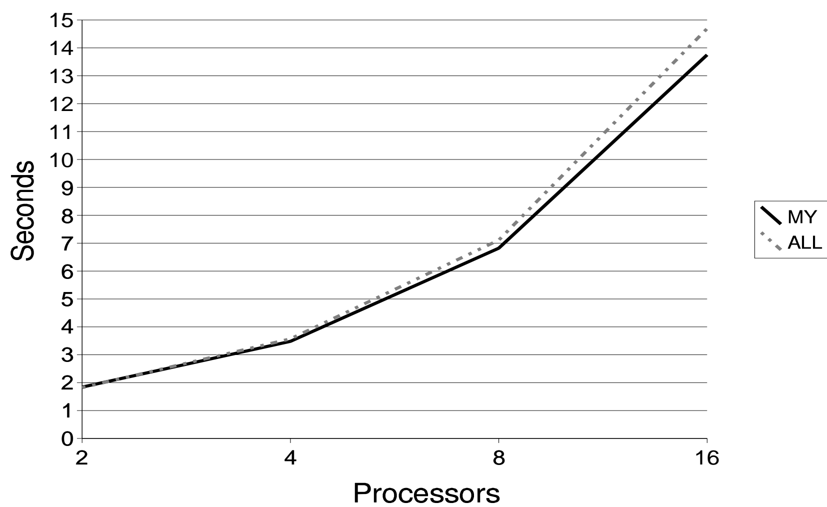


Figure 34: Flyer run times of push-based upc_all_scatter for 1000 iterations.



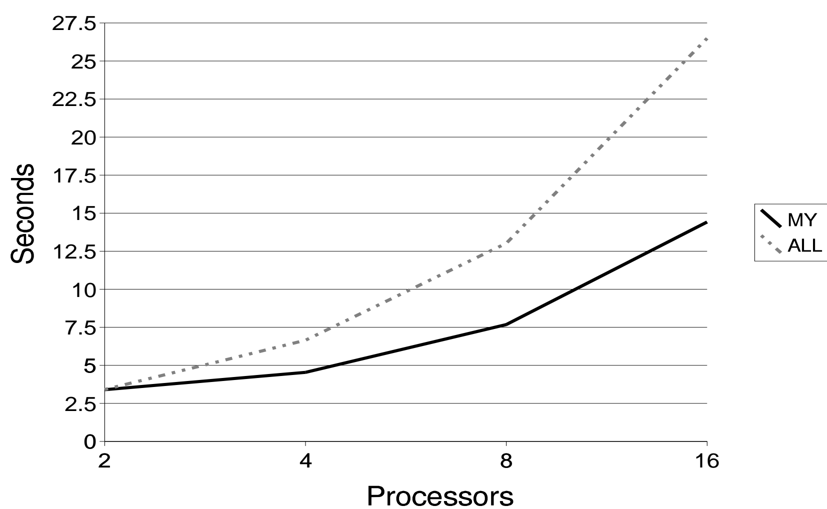
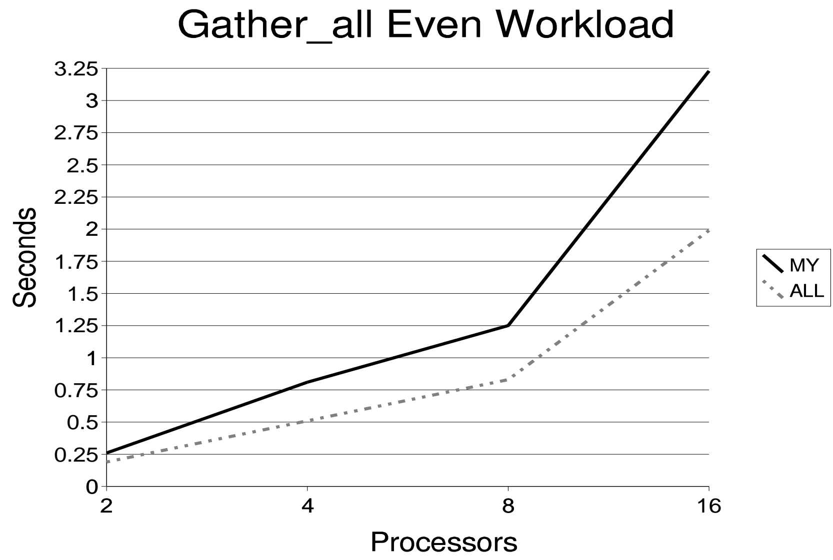


Figure 35: Flyer run times of a push-based upc_all_gather 1000 iterations.



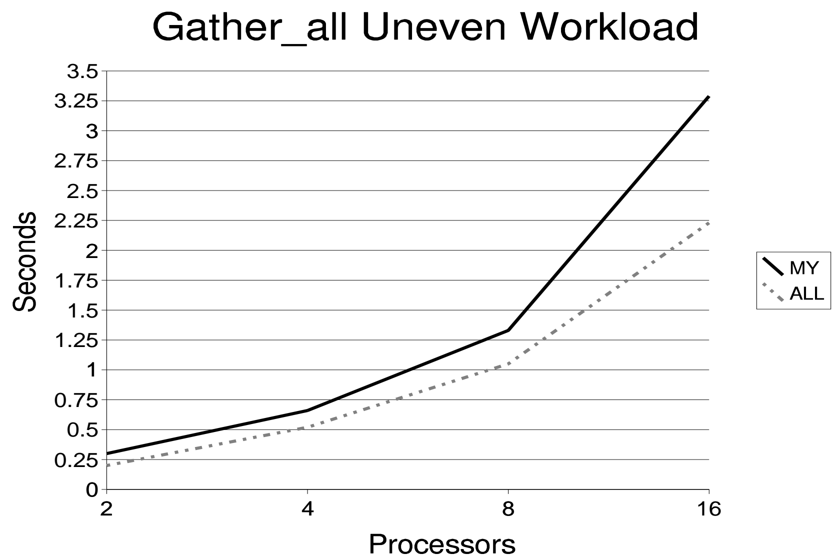
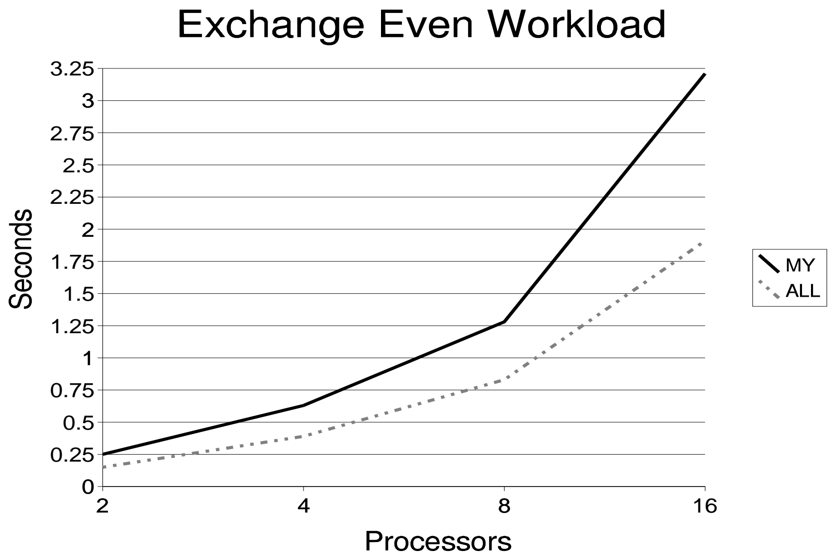


Figure 36: Flyer run times of a push-based upc_all_gather_all for 1000 iterations.



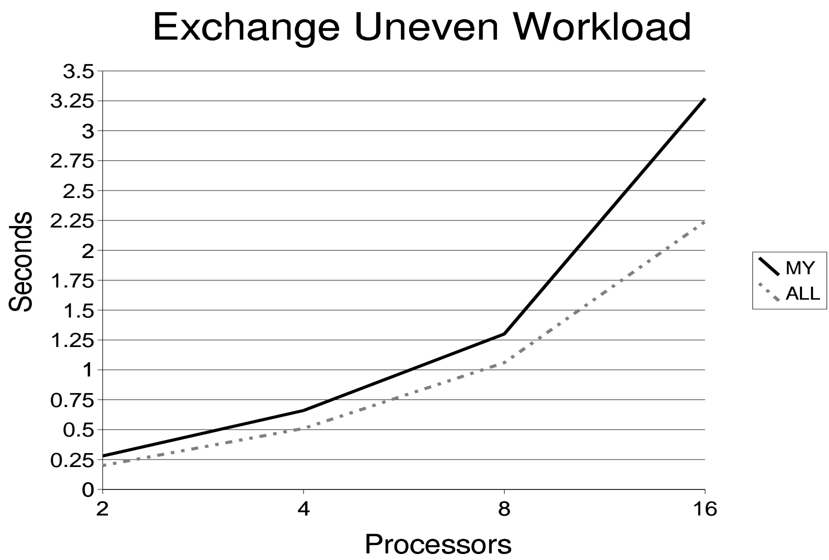
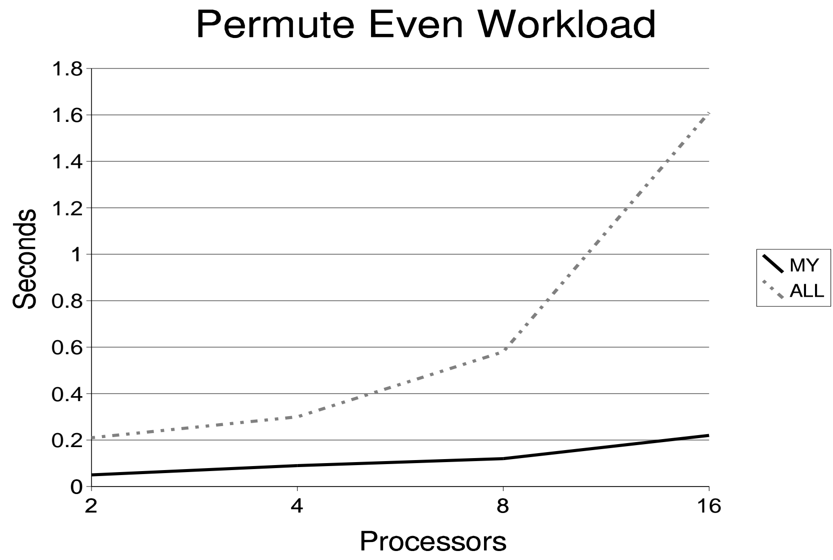


Figure 37: Flyer run times of a push-based upc_all_exchange for 1000 iterations.



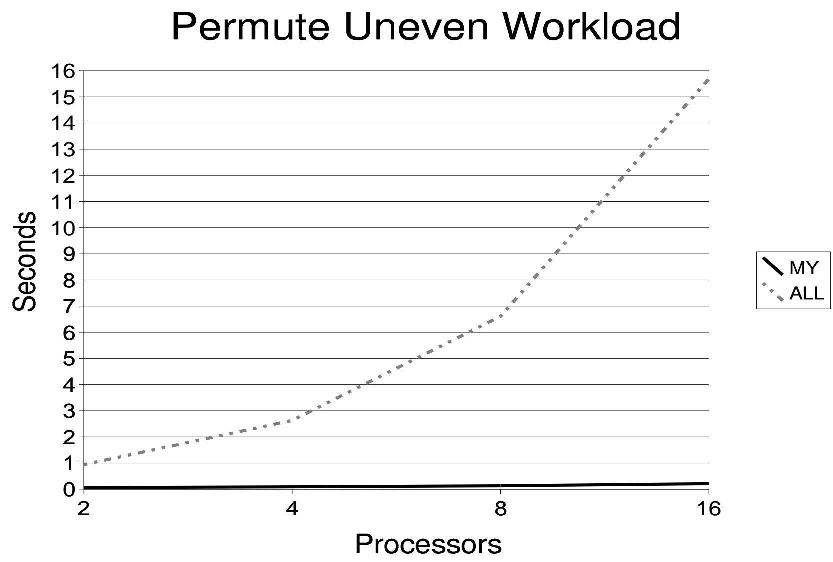


Figure 38: Flyer run times of a push-based upc_all_permute for 1000 iterations.

References

[1] T. El-Ghazawi, W. Carlson, and J. Draper. UPC language specifications, V1.1. Technical report, George Washington University and IDA Center for Computing Sciences, March 24, 2003. http://www.gwu.edu/ upc/docs/upc_spec_1.1.1.pdf

[2] E. Wiebel, D. Greenberg and S. Seidel. UPC Collectives Operations Specifications, Dec 2003. http://www.gwu.edu/ upc/docs/UPC_Coll_Spec_V1.0.pdf

[3] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C 1993 http://www.cs.berkeley.edu/projects/parallel/castle/split-c/split-c.tr.html

[4] B. Gorda, K. Warren and E. D. Brooks III, Programming in PCP, June, 1991 http://citeseer.nj.nec.com/330095.html

[5] R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *Fortran Forum*, volume 17, no 2, section 3, 1998

[6] P. N. Hilfinger (editor), D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, *Titanium Language Reference Manual*, Version 1.10, 4 November 2003 http://www.cs.berkeley.edu/Research/Projects/titanium/doc/lang-ref.pdf

[7] G. J. Holzman, The Model Checker Spin, *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.

[8] F. Petrini, D. J. Kerbyson, S. Pakin, *The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q*, 2004, http://www.sc-conference.org/sc2003/paperpdfs/pap301.pdf

[9] T. A. El-Ghazawi, W. W. Carlson, J. M. Draper, *UPC Language Specifications V1.1.1*, October 7, 2003 http://www.gwu.edu/ upc/docs/upc_spec_1.1.1.pdf