Computer Science Technical Report

A Specification of The Extensions to The Collective Operations of Unified Parallel C

Zinnu Ryne and Steven Seidel

Michigan Technological University
Computer Science Technical Report
CS-TR-05-08
2005

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

Abstract

Unified Parallel C (UPC) is an extension of the C programming language that provides a partitioned shared-memory model for parallel programming. On such platforms, a simple assignment operation can perform remote memory reads or writes. Nevertheless, since bulk transfers of data are more efficient than byte-level copying, collective operations are needed, and as a result a specification for a standard set of collective operations was proposed in 2003. This specification is now part of the UPC V1.2 language specification. However, these operations have limited functionality. This thesis work explores ways to overcome these limitations by proposing a set of extensions to the standard collective relocalization operations. The proposal includes performance and convenience improvements that provide asynchronous behavior, an in-place option, runtime implementation hints and subsetting feature. Also proposed are the new one-sided collective operations which are alternatives to the traditional MPI way of looking at collective operations.

A comparative performance analysis is done to ensure that the reference implementation of the extended collective operations perform within reasonable bounds of that of the standard collective operations. This provides necessary support to adopt these extensions as they provide greater flexibility and power to the programmer with a negligible performance penalty.

Contents

List of Figures

1 Introduction

1.1 Parallel Computing

Advances in microprocessor technology have revolutionized uniprocessor systems and yielded clock speeds of over 3.0 GHz. As a result, an observer may be inclined to surmise that such a system is capable of handling any computational need. However, applications such as weather forecasting, human genome sequencing, virtual reality, etc., continue to exceed the capabilities of such computer systems and require more computational speed than presently available in a uniprocessor system. It seems that whatever the computational speed of current processors may be, there will always be applications that require still more computational power. It is this demand for computational speed that leads to parallel computing.

The basic idea of parallel computing is simple: divide a big task into multiple smaller subtasks and execute them in parallel on different processing elements. The objective is to reduce the total time spent on the whole task.

Parallel computing platforms can be organized from two different perspectives: the physical organization (or the *architectural model*), which refers to the actual hardware organization of the platform, and logical organization (or the *programming model*), which refers to a programmer's view of the platform as provided by the language.

1.2 Architectural Model

From an architectural perspective there are two types of parallel platforms: shared memory and distributed memory.

1.2.1 Shared Memory Platforms

In a shared memory platform, there is a single physical memory that all processors have access to (Figure 1). The time taken by a processor to access any memory word in such a system is identical. As a result, this type of platform is also classifi ed as *uniform memory access (UMA)* multicomputer.

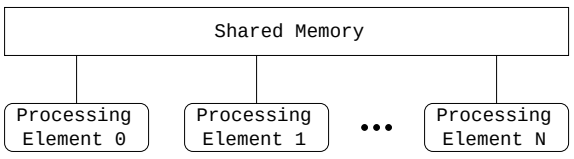


Figure 1: Shared memory architecture

1.2.2 Distributed Memory Platforms

In a distributed memory platform, each processor has its own private memory and an interconnection network (Figure 2). The time taken by a processor to access certain memory words in such a system is longer than others. As a result, this type of platform is also classified as *non-uniform memory access (NUMA)* multicomputer.

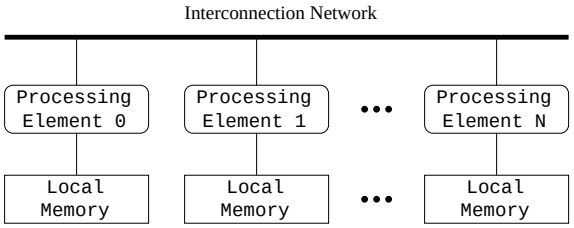


Figure 2: Distributed-memory architecture

1.3 Programming Model

From a programmer's perspective, there are two types of parallel platforms: shared address space and message passing.

1.3.1 Shared Address Space Platforms

The *shared address space* view of a parallel platform presents a common address space that is accessible to all processors. Processors interact by modifying data stored in this shared address space. This model is natural for shared memory systems (Figure 1). However, it can be implemented on distributed memory systems (Figure 2) through the use of a runtime system (RTS). The RTS emulates a shared address space architecture (Figure 3) on a distributed memory computer. Since accessing another processor's memory requires sending and receiving messages, this is costly. To cut the cost, algorithms exploit locality, structuring data and computation accordingly. The presence of a notion of global memory space makes programming in such platforms easier.

1.3.2 Message Passing Platforms

The logical machine view of a *message-passing* platform consists of multiple processing nodes, each with its own exclusive address space (Figure 2). On such platforms, interactions between processes running on different nodes are accomplished using interprocessor messages. This exchange of messages is used to transfer data and to synchronize actions among the processes. This model is natural for distributed memory architecture (Figure 2). However, emulating this model on a shared address space computer is not difficult. This

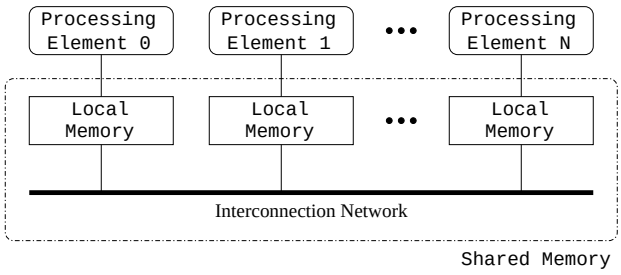


Figure 3: Distributed shared-memory architecture

can be done by partitioning the shared address space and assigning one equivalence class exclusively to each processor (assuming uniprocessor nodes).

The message-passing programming paradigm is one of the oldest and most widely used approaches for programming parallel computers. However, shared-address-space programming is gaining popularity, mainly because of programming simplicity.

In a multiprocessor system, each processor has a separate program. Each of these programs generate a single instruction stream for each processor, which operates on different data. This type of computer is known as *multiple instruction stream-multiple data stream (MIMD)*. Both the shared memory and the distributed memory multiprocessors fall in this category. A simple variant of this model, the *single program multiple data* (SPMD), is widely used. SPMD relies on multiple instances of the same program executing on different data. Examples of such parallel platforms are Sun Ultra Servers, multiprocessor PCs, workstation clusters, and IBM SP.

2 UPC

Unified Parallel C (UPC) is an extension of the C programming language designed for parallel programming. It provides a "partitioned" shared-memory model to the programmer. One advantage of such a model is that it allows for exploitation of locality and eliminates the need for message passing.

UPC uses an SPMD model of computation. The programmer sees the UPC program to be a collection of threads that share a common global address space. Each thread has its own private address space and part of the global address space which it can access as local. It is more efficient for a thread to access its own part of the global address space than some other thread's (Figure 4).

Any valid C program is a valid UPC program. In order to express parallelism, UPC extends ISO C 99 in several areas. It provides a "shared" type qualifier, which is similar to `const`, `volatile`, etc. It also carries a positive integer value with it that defines the shared block size. It describes the way in which the shared object is to be distributed over

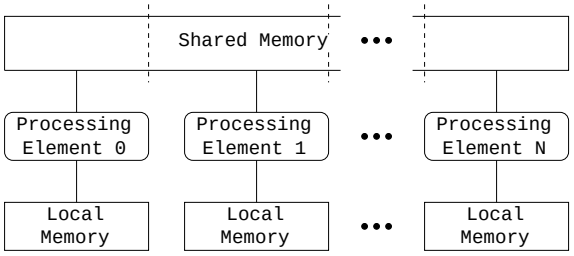


Figure 4: UPC's memory architecture

the threads.

The language defines two constants, `MYTHREAD` and `THREADS`. One is used to distinguish between the threads of an UPC program, and the other is to represent the amount of parallelism available to a program. The value of `THREADS` can be set at compile time or at run-time.

2.1 History

UPC is a product of many years of independent and collaborative work done by leading academic, industrial and government research groups. Immediate predecessors of UPC are *Parallel C Processor (PCP)*[9], developed by Brent Gorda, Karen Warren, Eugene D. Brooks III at the Livermore Lawrence National Laboratory in 1991., *Split-C*[7], developed by David E. Culler, Kathy Yelick et. al. at the University of California in 1993, and *AC*[6], developed by Jesse M. Draper and William Carlson at the IDA Supercomputing Research Center in 1995. All of these languages are parallel extensions to ISO C 99. UPC is a synthesis of these three languages (hence the name "unified"). For a comparative study of these languages and UPC, see [10].

Similar languages of current time are *Co-array Fortran*[5], a small extension of Fortran 95, and *Titanium*[14], an explicitly parallel dialect of Java, developed at University of California - Berkeley. All of these languages have an SPMD memory model.

2.2 Implementations

The first version of the UPC compiler was released in May 1999. Today there are several compilers available which includes GCC for the Cray T3E[2], HP's compiler for the Compaq AlphaServer[3], University of California-Berkeley's UPC[1], and Michigan Technological University's MuPC runtime system[4].

2.3 Terms and definitions

The following terms and definitions apply for the purpose of this document. Other terms are defined as footnotes where appropriate.

1. **Collective**:
   A constraint placed on some language operations which requires all threads to execute them in the same order. Any actual synchronization between threads need not be provided. The behavior of collective operations are undefined unless all threads execute the same sequence of collective operations.

2. **Thread**:
   According to the UPC Language Specification[12], a thread is "an instance of execution initiated by the execution environment at program startup". This means that processes in UPC are known as threads.

3. **Shared**:
   An area of memory in the shared address space. This area can be accessed by all threads. The keyword `shared` is used as a type qualifier in UPC.

4. **Private**:
   An area of memory in the local address space. Only the thread which owns the local address space can access it.

5. **Affinity**:
   A logical association between shared objects and threads. Each element of data storage that contains shared objects has affinity to exactly one thread.

6. **Phase**:
   An unsigned integer value associated with a pointer-to-shared which indicates the element-offset within a block of shared memory. Phase is used in pointer-to-shared arithmetic to determine affinity boundaries.

7. **Pointer-to-shared**:
   A local pointer pointing to a shared object.

8. **Pointer-to-local**:
   A local pointer pointing to a local object.

9. **Single valued**:
   An argument to a collective function which has the same value on every thread. The behavior of the function is otherwise undefined.

2.4 Language Basics

Since UPC is an extension to C, any C program is intuitively a valid UPC program. Therefore, all of C's syntax and semantics apply along with few additions. A minimal explanation of the UPC language constructs tailored to this document is presented here. For details on programming in UPC, see [8, 12].

2.4.1 Predefined identifiers

`THREADS` and `MYTHREAD` are expressions with values of type *int*. The first one specifies the number of threads and has the same value on every thread, whereas the second one specifies a unique thread index (`0...THREADS-1`). In figure 5, `THREADS` is 3 and `MYTHREAD` values are 0, 1 and 2 (labeled under each thread).

2.4.2 Data declaration

Data that is local to a thread is declared the same way as C; however, shared data is explicitly declared with a *type qualifier*, `shared`. For example, a shared variable is declared as follows:

```
shared int num;
```

Here `num` is of type *int*. The `shared` keyword is prepended to explicitly state that the data stored in this variable resides in shared address space and is accessible by all threads. (Figure 5)

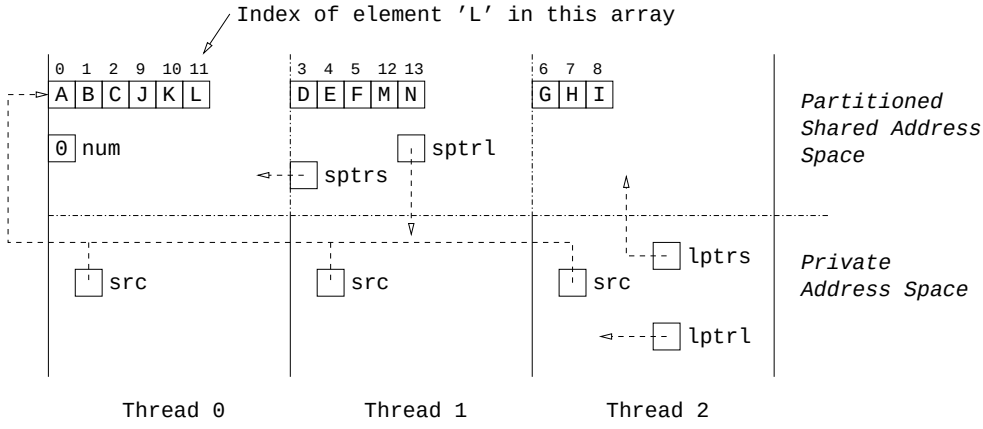


Figure 5: A visual representation of UPC's memory layout

2.4.3 Arrays

UPC provides the ability to explicitly lay out arrays in shared address space across threads. For example, an array in shared address space is declared as follows:

```
shared [3] char src[14];
```

Here `src` is a character array of length 14. The `shared` keyword is proceeded by a *layout qualifier*. Its purpose is to block the array in chunks of 3 in round-robin fashion. As a result, the array's first 3 elements are in thread 0, the next in thread 1 and so on, and when it hits the last thread, it winds up to the first thread and keeps on going until all the elements are placed (Figure 5).

The number in the layout qualifier is known as the *blocking factor* or *block size* of the array. It can be any positive integer, not specified (i.e., `[]`), or `*`. If the number is `0` or is not specified then this indicates an *indefinite* blocking factor (i.e., all elements have affinity to the same thread). The `[*]` indicates that the shared array is distributed equally in order. If there is no layout qualifier at all, then the block size has the default value, which is `1`.

Note that the block size is part of the type compatibility. Therefore, type casting between types with different block sizes is illegal.

2.4.4 Pointers

UPC has four types of pointers:

1. (local) pointer to local, declared as
   ```
   char * lptrl;
   ```
2. (local) pointer to shared, declared as
   ```
   shared char * lptrs;
   ```
3. shared pointer to shared, declared as
   ```
   shared char shared * sptrs;
   ```
4. shared pointer to local, declared as
   ```
   char shared * sptrl;
   ```

Note that a shared pointer to local can only point to local data of a thread to which it has affinity to, not any others'. (Figure5)

2.4.5 Iteration constructs

Besides all of C's iteration constructs, UPC provides `upc_forall`. Its syntax extends C's `for` construct by adding a fourth field after the increment expression. This field denotes affinity.

In the following code segment, `upc_forall` is used to print out contents of the shared array `src` along with the thread number which prints it.

```
int i;
upc_forall( i = 0; i < 14; i++; &src[i] )
{
   printf( "Th = %d. %c\n", MYTHREAD, src[i] );
}
```

Each thread only prints out data that is in its partition of the shared address space (i.e., data that has affi nity to that particular thread). Therefore, thread `1` will only print out `D`, `E`, `F`, `M`, and `N`. (Figure 5)

Note that `upc_forall` is a collective operation in which, for each execution of the loop body, the controlling expression and affi nity expression are single valued.

2.4.6 Thread synchronization

Since threads in UPC communicate by accessing shared data, synchronizing these accesses is vital to the proper functioning of parallel programs. To achieve this, UPC provides `upc_barrier`. When `upc_barrier` is called, all threads wait for all other threads to synchronize at that point. Once all threads have reached that particular point, they can advance in program execution.

2.4.7 Bulk transfers

UPC provides `upc_memcpy` which is analogous to C's `memcpy`. Basically, it copies a chunk of shared memory area from one place to another. However, one restriction is that the shared memory area may not cross a thread boundary.

3 Standard Collective Operations

3.1 Background

A collective operation is a construct which requires all processing elements to execute a certain function. In a message passing programming model, collective operations play an important role in relocating data, performing computation, and synchronizing processes.

It could be argued that if UPC is truly based on a shared memory model, why is there a need for collective operations? The answer can be found within the two aspects of parallel programming - performance and programmability. Bulk transfer of data is usually more effi cient than element-by-element transfers. For the purpose of elaboration, consider the collective operation *broadcast*. In this collective operation one thread has a block of data

that needs to be copied to all other threads. The programmer may choose to use a collective function or individually assign memory locations to copy over the required data. If the required data is a string, each thread would have to go through a loop to collect each character in the string, one at a time. Although all the threads would accomplish the task simultaneously, it would be inefficient. On the other hand, a collective call could just copy a block of data all at once, eliminating the performance degradation caused by the loop in each thread. For a small number of threads and a short string, this problem may seem irrelevant. However, as the number of threads and the length of the string grows, individual assignments become very inefficient. Furthermore, it is very natural for a programmer to want to perform a broadcast operation more than once in a program. In such case, it is convenient to have a standard function for this operation. Library writers can provide very efficient low-level implementations of collective operations by taking advantage of the underlying system architecture. Lastly, in the parallel programming domain, programmers are already familiar with collective operations in message passing models, such as MPI.

3.2 UPC's Standard Collective Relocalization Operations

The *UPC Collective Operations Specification V1.0*[13, 8] was released on December 12, 2003. At present, the document is part of the *UPC Language Specifications V1.2* [12] with the exception of the *sort* function, which has been deprecated. Details of all the relocalization and computational operations can be found in any of the aforementioned documents. However, in this section we present brief descriptions of the six *relocalization collectives*, which copy shared data from one location to another in the shared address space, and propose possible extensions to them in the following section.

For the following collective functions, in the synopses, `dst` and `src` are *pointers-to-shared* representing, respectively, the destination and the source of the data that is moved, and `nbytes` is the number of bytes to be copied from source to destination. For `upc_all_permute`, the shared permutation array is pointed to by `perm`. The `mode` argument specifies the synchronization mechanism.

In the figures, `A` is the source array and `B` is the destination array. The contents of `B` indicate the effect of the operation.

3.2.1 The `upc_all_broadcast` function

Synopsis:

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

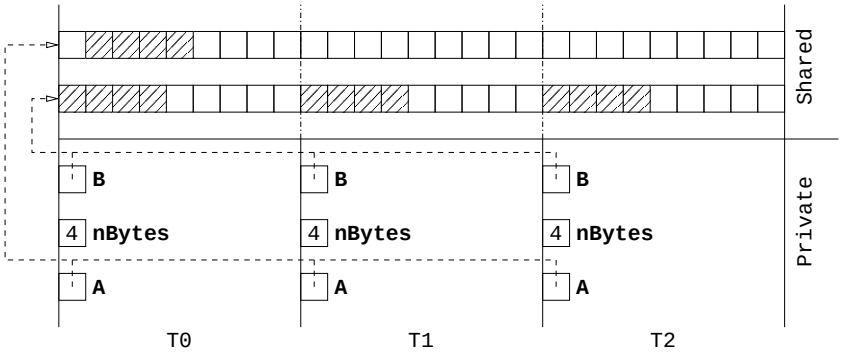


Figure 6: upc_all_broadcast

Brief description:

The upc_all_broadcast function copies a block of shared memory with affi nity to a single thread to a block of shared memory on each thread. The number of bytes in each block is nbytes, which must be strictly greater than 0.

3.2.2 The upc_all_scatter function

Synopsis:

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

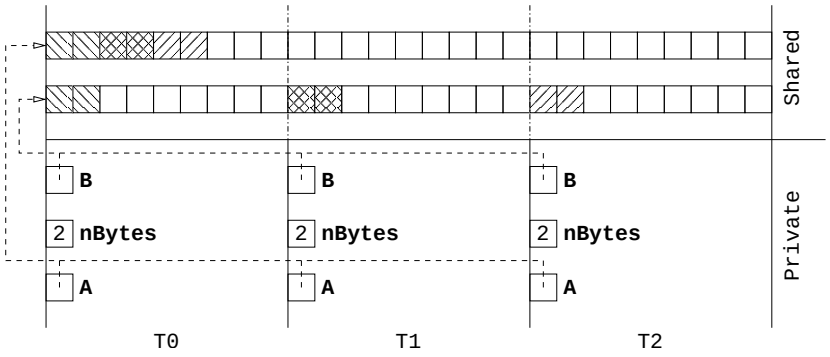


Figure 7: upc_all_scatter

Brief description:

The `upc_all_scatter` function copies the *i*th block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the *i*th thread. The number of bytes in each block is `nbytes`, which must be strictly greater than 0.

3.2.3 The `upc_all_gather` function

Synopsis:

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

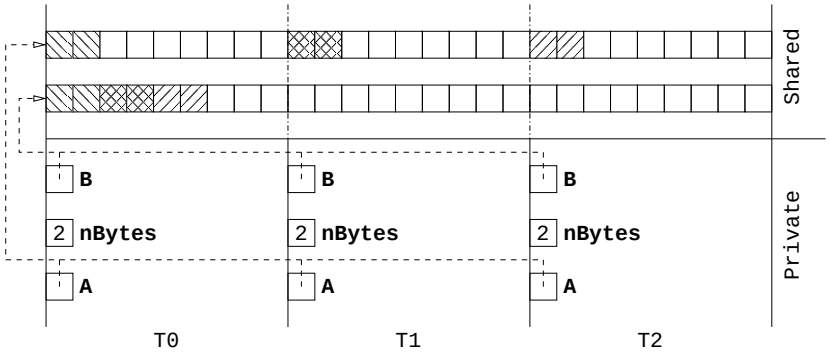


Figure 8: `upc_all_gather`

Brief description:

The `upc_all_gather` function copies a block of shared memory that has affinity to the *i*th thread to the *i*th block of shared memory area that has affinity to a single thread. The number of bytes in each block is `nbytes`, which must be strictly greater than 0.

3.2.4 The `upc_all_gather_all` function

Synopsis:

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

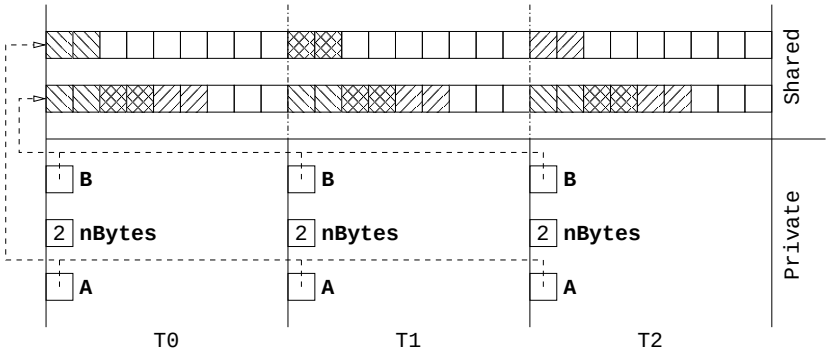


Figure 9: `upc_all_gather_all`

Brief description:

The `upc_all_gather_all` function copies a block of shared memory that has affi nity to the *i*th thread to the *i*th block of shared memory area on each thread. The number of bytes in each block is `nbytes`, which must be strictly greater than 0. This function is analogous to all the threads calling `upc_all_gather`.

3.2.5 The `upc_all_exchange` function

Synopsis:

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

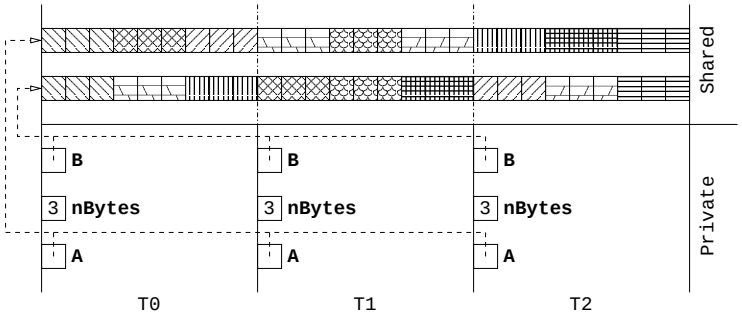


Figure 10: `upc_all_exchange`

Brief description:

The `upc_all_exchange` function copies the *i*th block of memory from a shared memory area that has affinity to thread *j* to the *j*th block of a shared memory area that has affinity to thread *i*. The number of bytes in each block is `nbytes`, which must be strictly greater than 0.

3.2.6 The `upc_all_permute` function

Synopsis:

```
void upc_all_permute( shared void * restrict dst,
                      shared const void * restrict src,
                      shared const void * restrict perm,
                      size_t nbytes,
                      upc_flag_t mode );
```

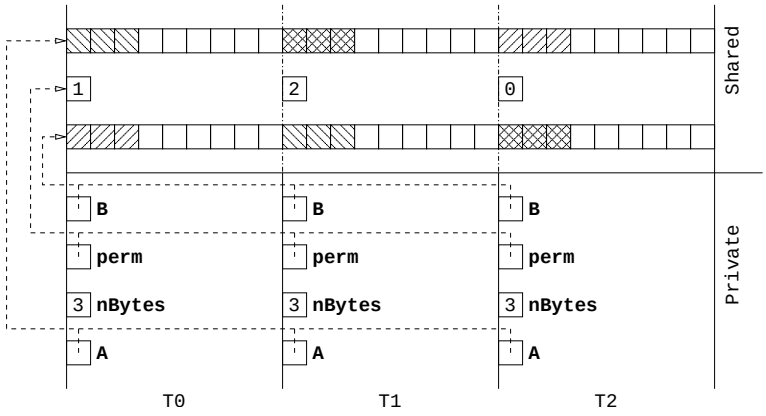


Figure 11: `upc_all_permute`

Brief description:

The `upc_all_permute` function copies a block of shared memory area that has affinity to the *i*th thread to a block of shared memory area that has affinity to thread `perm[i]`. The number of bytes in each block is `nbytes`, which must be strictly greater than 0. `perm[0..THREADS-1]` must contain `THREADS` distinct values: 0, 1, ..., `THREADS`-1.

3.3 Limitations of the Standard Relocalization Collective Operations

The standard relocalization collectives are limited in their features. Since the `nbytes` argument is single valued over all threads, the size of data movement is the same. However, there could be cases in functions other than `upc_all_broadcast` where a varying size is desired. For example, a variable length *exchange* operation is used in parallel bucket sort. With the current set of collective operations, this is not possible.

For `upc_all_gather` and `upc_all_gather_all`, the assumption is that the source data resides in the beginning block of each thread. For `upc_all_broadcast` and `upc_all_scatter`, the destination data is copied to the beginning block of each thread. This is true for both the source and destination data of `upc_all_exchange` and `upc_all_permute`. The current specifications do not allow any flexibility in the placement of data.

The data blocks of the source array of `upc_all_broadcast` and `upc_all_scatter`, and of the destination array of `upc_all_gather` and `upc_all_gather_all` are contiguous. This is true for both the source and destination arrays of `upc_all_exchange`. The current specifications do not allow specifying strides in data or explicitly specifying data locations.

All of these collective operations assume that all of the source or destination data-blocks reside in a single array (source data-blocks in a single source array, and destination data-blocks in a single destination array). The specifications do not, for example, allow data to be scattered in different arrays in the destination threads.

In all these operations source thread(s) copy data to destination thread(s). It could be the case that the source and destination are in the same thread. This could be undesirable as it requires unnecessary copying. For example, a user wishing to do a matrix transpose can use `upc_all_exchange`; but for that we need another array with the same type and size. After the operation, the new structure would contain the resulting matrix. The current specifications provide no way to perform such operations "in place". We cannot simply have `src` and `dst` be pointers to the same location as that would violate the specification because of the `restrict` modifier.

The reference implementations of these collective operations can be compiled as either a "pull" or a "push" implementation. In a "pull" implementation, destination thread(s) pull(s) data from source thread(s), whereas in the "push" implementation source thread(s) push data to the destination thread(s). The programmer has no way of choosing the implementation at runtime.

Finally, once a collective function is issued, all threads participate in it. There is no way to perform the collective operations only on a subset of threads.

4 Extensions to the Collective Operations

4.1 Background

The limitations of the standard collective relocalization operations justify the need for extended versions which provide more flexibility to the programmer.

The MPI library provides a set of collective operations [11] as well as their variants. The proposed extensions[16] for UPC closely follow MPI's approach. However, there are subtle differences in the programming models, which naturally translates to syntax differences. It is important to note that MPI does not provide anything analogous to UPC's permute operation; nor is synchronization as big of an issue with collective operations in MPI as it is in UPC.

For each standard UPC collective function (see 3.2), there can be two variants. In the "vector" variant, each block of data can be of different size; and it allows the function to pick distinct non-contiguous data-blocks. The second variant is a further generalization of the first variant. In this case, the programmer may explicitly specify each data-block and their size.

The specification design phase of these functions was driven by the question: "what does a thread know?". In other words, what information does each thread require to simultaneously and independently (as much as possible) perform the operation taking advantage of locality? Another aspect that motivated the specification was "how does the programmer perceive the programming model?". For example, a programmer may declare an array of integers, say `A`, and specify the second element in the array using normal array indexing, `A[1]`. However, since UPC handles everything at the byte-level[1], the second element would actually be 4 bytes (assuming integers are 4 bytes) away from the first one. Therefore, the goal is to free the programmer from the hassle of calculating these byte addresses which can be done internally with little effort, provided that the data-type is known.

There are several ways in which the "in place" option can be implemented. One straight-forward way is to compare the `src` and `dst` pointers and see if `src == dst`. Another approach is to specify a value in the *mode* argument which would trigger the *in place* behavior and ignore either `src` or `dst`. However, both of these approaches violate the `restrict` and `const` modifiers. A proposal from UC-Berkeley pointed out that passing the defined type `UPC_IN_PLACE` as the `src` argument could solve this issue without violating any of the existing specifications. This works for `upc_all_exchange`, `upc_all_permute` and `upc_all_gather_all`, but not so easily for `upc_all_broadcast`, `upc_all_scatter` and `upc_all_gather`. The data layouts of `src` and `dst` in these collective functions make it harder to implement the "in place" option in such fashion.

[1] If we consider integers to be of 4 bytes, then the first element of an integer array would be at byte location 0, the second one at byte location 4, the third one at byte location 8, and so on. UPC programmers may access array elements with simple array indexing, but internally this is traslated to the byte addressing form.

Users can choose either “push” or “pull” implementation of the operations at runtime. The constants UPC_PUSH and UPC_PULL have been defined, which can be bitwise or-ed with other defined constants in the *mode* argument.

4.2 Proposed Extensions

Presented here are variants of three of the *relocalization collectives* - broadcast, scatter and exchange. For details on these and other collective operations, see [16].

In the synopses for the vector variants of the following functions, src and dst are *pointers-to-shared* representing, respectively, the source and destination of the data that is moved. sdisp and ddisp are *pointers-to-shared* representing respectively, arrays of displacements of the source and destination data. src_blk and dst_blk are single valued arguments which represent, respectively, the block sizes of source and destination arrays. nelems is a pointer to a shared array each of whose elements specify the number of elements to be copied from source to destination. The typesize argument specifies the size of the type. The mode argument specifies the synchronization mechanism. In the synopses for the generalized functions, src and dst are pointers for shared arrays each of whose elements point to a shared memory area. Except in broadcast, nbytes is a shared array of THREADS sizes, each representing size of data transfer for that particular thread.

Depending on the presence of UPC_ASYNC in the *mode* argument, the functions immediately return a *handle* of type upc_coll_handle_t to be consumed by a completion function at a later time. The synchronization modes are respected in the corresponding completion function.

In the figures, A is the source array and B is the destination array. The contents of B reflect the effect of the operation.

4.2.1 Broadcast: Vector Variant

For the first of the two variants of the standard broadcast operation, we intend to be able to copy the data-block from the source array to the destination array at specific locations in each thread.

Synopsis:

```
void upc_all_broadcast_v( shared void * dst,
                          shared const void * src,
                          shared size_t * ddisp,
                          size_t nelems,
                          size_t dst_blk,
                          size_t typesize,
                          upc_flag_t mode );
```

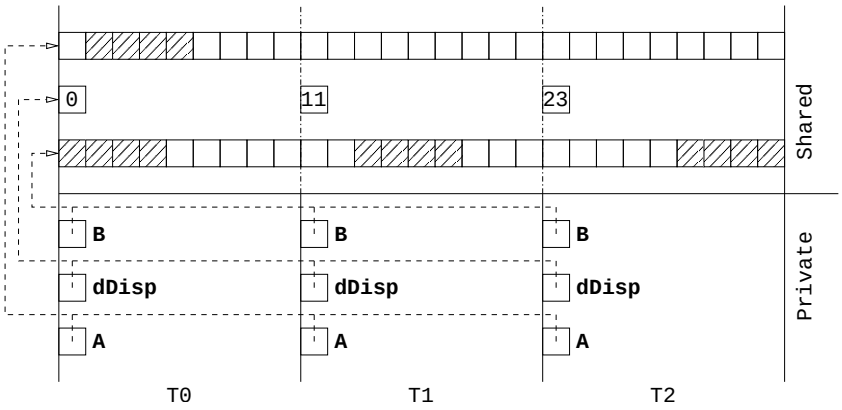


Figure 12: `upc_all_broadcast_v`

Brief Description:

The flexibility as to where the data is copied to on each destination thread is achieved with the help of the new argument `ddisp`. It is a pointer to a shared array with the type `shared [1] size_t [THREADS]`. The programmer provides the array index in `ddisp[i]` to specify where in thread *i* the data is to be placed. `nelems` specifies the number of elements to broadcast. In order for the function to work as intended, this value must be strictly greater than 0. `dst_blk` is the blocking factor for the destination array. Since the destination pointer is passed in as argument casted as `shared void *`, the blocking information is lost, which is needed to perform address calculation. `typesize` is the size of the data-type.

In summary, `upc_all_broadcast_v` copies `nelems` elements of the shared array pointed to by `src` having affinity to a single thread to a block of shared memory pointed to by `dst` and with `ddisp[i]` displacement on thread *i* (Figure 12).

See Appendix B for a reference implementation.

4.2.2 Broadcast: Generalized Function

This is the most general of broadcast functions. The user is expected to provide explicit addresses for the source and each destination. As a result, the destinations for each thread can be in separate shared arrays.

Synopsis

```
void
upc_all_broadcast_x( shared void * shared * restrict dst,
                     shared const void * restrict src,
```

```
                        size_t nbytes,
                        upc_flag_t mode,
                        upc_team_t team );

upc_coll_handle_t
upc_all_broadcast_x( shared void * shared * restrict dst,
                        shared const void * restrict src,
                        size_t nbytes,
                        upc_flag_t mode,
                        upc_team_t team );
```

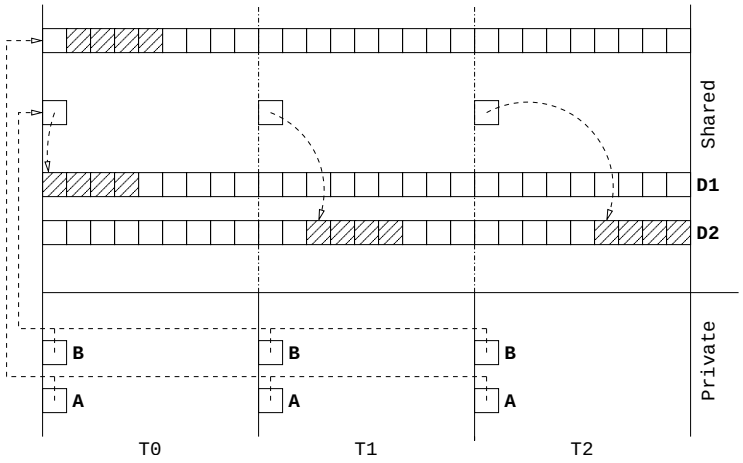


Figure 13: `upc_all_broadcast_x`

Brief Description

This function allows the flexibility of broadcasting data to separate arrays. As a result the `dst` pointer is now treated as pointing to a shared memory area with type `shared [] char * shared [THREADS]`. In plain terms, `dst` points to a shared array, each of whose elements point to an area in shared memory.

In summary, `upc_all_broadcast_x` copies `nbytes` bytes of the shared array pointed to by `src` having affi nity to a single thread to the areas of shared memory pointed to by `dst[i]` on thread *i* (Figure 13).

See Appendix B for a reference implementation.

4.2.3 Scatter: Vector Variant

For the first of the two variants of the standard scatter operation, we intend to be able to scatter non-contiguous blocks of varying sizes from the shared array on the source thread to specific locations in the destination array on each thread.

Synopsis:

```
void upc_all_scatter_v( shared void * dst,
                        shared const void * src,
                        shared size_t * ddisp,
                        shared size_t * sdisp,
                        shared size_t * nelems,
                        size_t dst_blk,
                        size_t src_blk,
                        size_t typesize,
                        upc_flag_t mode );
```

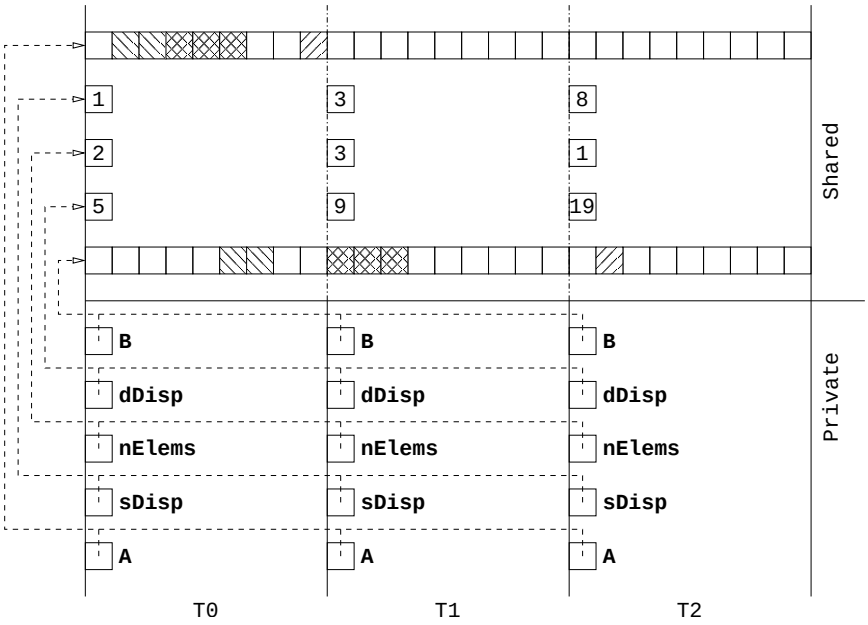


Figure 14: `upc_all_scatter_v`

Brief Description:

The flexibility as to where the data is copied from on the source thread and copied to on each destination thread is achieved with the help of the new arguments `sdisp` and `ddisp`. These are pointers to shared arrays with the type `shared [1] size_t [THREADS]`. `sdisp[i]` contains the index of the *i*th block in the source array and `ddisp[i]` contains the index of where in the destination array in thread *i* the data is to be placed. `nelems` is a pointer to shared array with type `shared [1] size_t [THREADS]`. `nelems[i]` specifies the number of elements to copy to thread *i*. `src_blk` and `dst_blk` are blocking factors for the source and destination arrays respectively. These are needed to perform address calculation because the blocking information is lost due to typecasting as `shared void *` in parameter list. `typesize` is the size of the data type.

In summary, `upc_all_scatter_v` copies the *i*th block of `nelems[i]` elements of the shared array pointed to by `src` at displacement `sdisp[i]` and with affinity to a single thread to a block of shared memory pointed to by `dst` and with `ddisp[i]` displacement on thread *i* (Figure 14).

4.2.4 Scatter: Generalized Function

This is the most general of the scatter functions. The user is expected to provide explicit addresses for each block of data. As a result, the data blocks in the source thread and destinations for each thread can be in separate arrays.

Synopsis:

```
void
upc_all_scatter_x( shared void * shared * restrict dst,
                   shared const void * shared * restrict src,
                   shared size_t * restrict nbytes,
                   upc_flag_t mode,
                   upc_team_t team );

upc_coll_handle_t
upc_all_scatter_x( shared void * shared * restrict dst,
                   shared const void * shared * restrict src,
                   shared size_t * restrict nbytes,
                   upc_flag_t mode,
                   upc_team_t team );
```

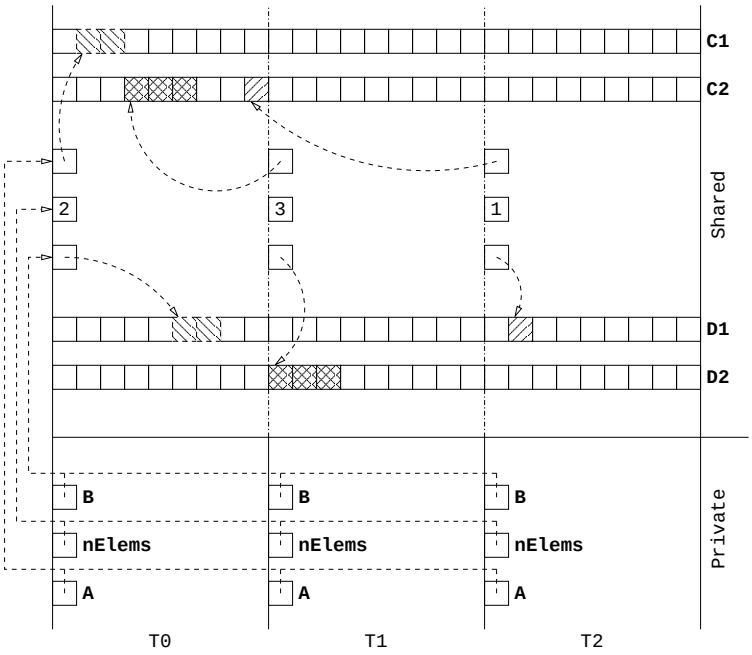


Figure 15: `upc_all_scatter_x`

Brief Description:

This function allows the flexibility of copying data-blocks from separate arrays on the source thread to separate arrays in each destination thread. As a result the `src` and `dst` pointers are both treated as pointing to a shared memory area with type `shared [] char * shared [THREADS]`. In simple terms, both `src` and `dst` point to shared arrays, each of whose elements point to an area in shared memory.

In summary, `upc_all_scatter_x` copies `nbytes[i]` bytes of the shared memory pointed to by `src[i]` on the source thread to a block of shared memory pointed to by `dst[i]` on thread *i* (Figure 15).

4.2.5 Exchange: Vector Variant

For the first of the two variants of the standard exchange operation, we intend to be able to exchange non-contiguous blocks of varying sizes among threads.

Synopsis:

```
void upc_all_exchange_v( shared void * dst,
                         shared const void * src,
                         shared size_t * ddisp,
```

```
shared size_t * sdisp,
shared size_t * nelems,
size_t src_blk,
size_t dst_blk,
size_t typesize,
upc_flag_t mode );
```

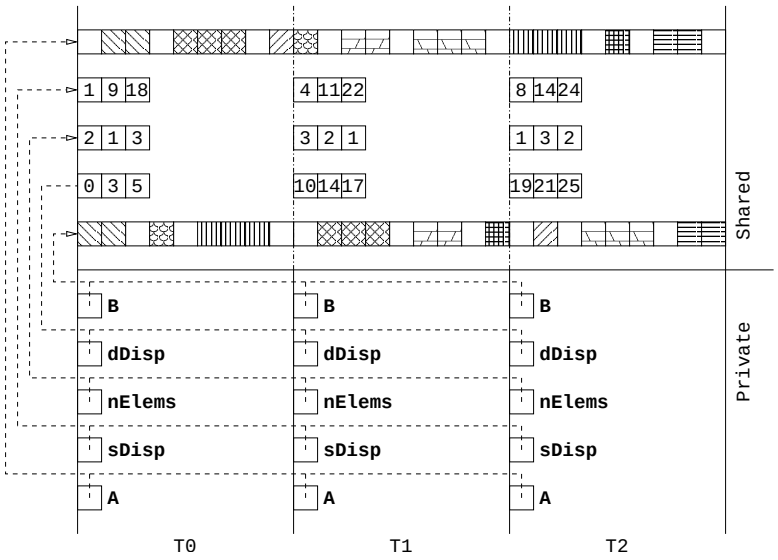


Figure 16: `upc_all_exchange_v`

Brief Description:

The flexibility as to where the data is copied from on the source threads and copied to on the destination threads is achieved with the help of the new arguments `sdisp` and `ddisp`. These are pointers to shared arrays with the type `shared [3] size_t [THREADS]`. `sdisp[i]` contains the index of the *i*th block in the source array and `ddisp[i]` contains the index of where in the destination array the data is to be placed. `nelems` is a pointer to shared array with type: `shared [3] size_t [THREADS]`. `nelems[i]` specifies the number of elements in the *i*th block in the source thread. `src_blk` and `dst_blk` are blocking factors for the source and destination arrays respectively. These are needed to perform address calculation because the blocking information is lost due to typecasting as `shared void *` in parameter list. `typesize` is the size of the data type.

In summary, `upc_all_exchange_v` copies the *i*th block of data in thread *j* to the *j*th block of memory in thread *i*. The sizes of the data blocks can be different and the data can be non-contiguous in both the source and destination arrays (Figure 16).

4.2.6 Exchange: Generalized Function

This is the most general of the exchange functions. The user is expected to provide explicit addresses for all memory areas. As a result, the data blocks in the source and destination arrays in each thread can be in separate arrays.

Synopsis:

```
void
upc_all_exchange_x( shared void * shared * restrict dst,
                    shared const void * shared * restrict src,
                    shared size_t * restrict nbytes,
                    upc_flag_t mode,
                    upc_team_t team );

upc_coll_handle_t
upc_all_exchange_x( shared void * shared * restrict dst,
                    shared const void * shared * restrict src,
                    shared size_t * restrict nbytes,
                    upc_flag_t mode,
                    upc_team_t team );
```

Brief Description:

This function allows the flexibility of copying data blocks from separate arrays on the source threads to separate arrays in the destination threads. As a result the `src` and `dst` pointers are both treated as pointing to a shared memory area with type `shared [] char * shared [THREADS*THREADS]`. In simple terms, both `src` and `dst` point to shared arrays, each of whose elements point to an area in shared memory. The other parameters are same as the vector variant.

In summary, `upc_all_exchange_x` copies `nbytes[i]` bytes of the shared memory pointed to by `src[i]` on source thread *j* to a block of shared memory pointed to by `dst[j]` on thread *i* (Figure 17).

4.3 Alternative Designs

In the vector variants of both *broadcast* and *exchange*, `ddisp` can be treated in two ways: starting from `ddisp` of thread 0, or starting from the beginning block of each thread. Although we have taken the first approach, the second one can also be taken. We have reasons to believe that both will yield the same performance. The only factor here that

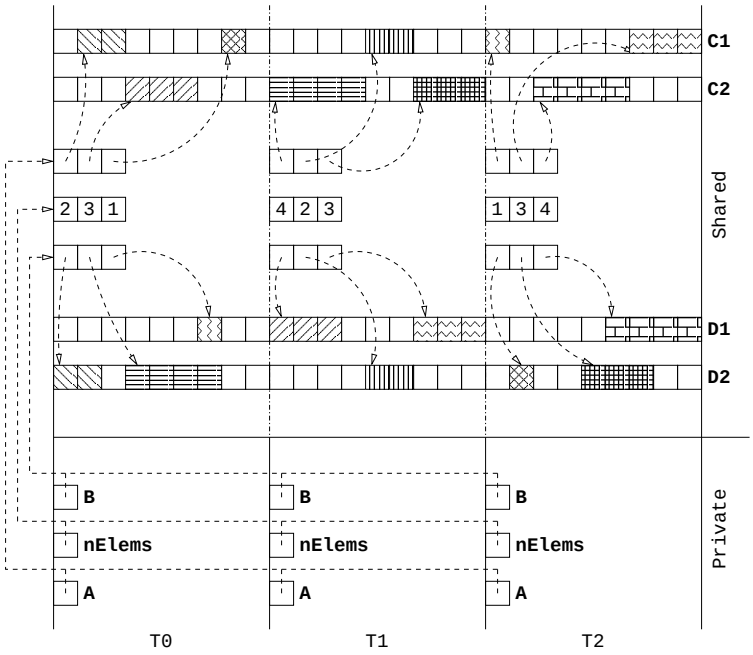


Figure 17: upc_all_exchange_x

led us to take the first approach is programmer convenience and familiarity with MPI's collective extensions.

In the standard collective operations of UPC, the size of data movement is specified in bytes. In the vector variants, the functions require number of elements and the size of the data type to be specified. This provides user convenience (see 4.1). Alternatively, we can require that the argument be in bytes, just as in the standard collective functions, and the data type be provided instead of its size. In that case, we will have to define constants, like UPC_INT, UPC_CHAR, etc., to be passed in as arguments.

4.4 Completion Operations

The asynchronous collective operations are two folded: posting operation and completion operation. The posting operation returns a *handle* of type upc_coll_handle_t immediately. This handle is then passed to a completion operation to guarantee completion of the operation.

```
void upc_wait( upc_coll_handle_t handle );
int upc_test( upc_coll_handle_t handle );
int upc_get_status( upc_coll_handle_t handle );
```

The upc_wait function blocks until the operation corresponding to the handle is complete. Once it is complete, the handle is set to UPC_HANDLE_COMPLETE so that any future references to this handle will have the information. upc_test on the other hand, returns whether the corresponding operation is complete or not. If it is complete, it does the same as upc_wait and returns a positive integer. Otherwise, it returns a negative integer. The upc_get_status is a convenience function to check whether a handle has been completed. It has no effect on the handle.

5 One-sided Collective Operations

Traditionally, collective operations have been thought of as two-sided memory operations. Such operations requires the processing elements (*i.e.*, threads) to synchronize their execution in order to guarantee certain semantics of the language. At the 2005 UPC workshop, there were mixed reactions on the subject of the aforementioned complex collective operations. A growing interest for functions that "just do" a collective operation resulted in a different stream of thought: "Can collectives be implemented as one-sided memory operations, like upc_global_alloc or upc_free?"

The reference implementations of the collective functions are implemented using UPC's upc_memcpy family of functions (*i.e.*, upc_memget, upc_memput, or upc_memcpy). When a call is made, such as upc_all_broadcast, after handling all synchronization issues, depending on what implementation hint (*i.e.*, "pull" or "push") is provided, threads call upc_memcpy to perform the actual data relocation. A careful observation reveals that upc_memcpy does not require the caller to be either the source or the destination of the relocalization operation. Therefore, a slightly more complex form of upc_memcpy can provide the compiler or the run-time system (RTS) with just enough information to be able to perform operations analogous to the collective operations.

For convenience, the following four relocalization operations have been proposed. More operations and detail specifi cation can be obtained from [15]. Since these do not fall in the traditional realm of collective operations, the _all_ part from the names have been carefully omitted to avoid any confusion.

5.1 The API

```
void upc_broadcast( shared void *dst,
                    shared const void *src, size_t nbytes )
void upc_scatter( shared void *dst,
                  shared const void *src, size_t nbytes )
void upc_gather( shared void *dst,
                 shared const void *src, size_t nbytes )
```

```
void upc_exchange( shared void *dst,
                   shared const void *src, size_t nbytes )
```

Gather-all can be implemented by all threads calling `upc_gather`. Because how it is implemented, permute operation can be simply handled by `memcpy`; in other words, there is no added advantage to have a separate function in this regard.

5.2 Cost/Benefit Analysis

Only a single thread needs to call the one-sided collective functions. When the call returns, it is safe to write to the *source* on the root thread. On all other threads, the copies of the source data will be available at the beginning of the next synchronization phase[2]; however, they will be observable by the calling thread after it returns from the call. Moreover, by nature these operations are asynchronous, so better performance is expected.

6 Performance Analysis

The performance study of this project involves a comparative analysis of the standard and extended collective operations. The reference implementations of both are implemented at the user-level with `upc_memcpy`; thus, the only difference expected is due to the extra level of pointer redirection in the extended versions.

The reference implementation of the asynchronous versions of the extended collective operations have been implemented as blocking. Therefore, their performance yield is same as the synchronous versions. The completion functions have been implemented to do nothing but return immediately.

6.1 Test Platforms

The reference implementation of both the standard and extended collective operations have been tested on a 20 2-way 2.0 GHz Pentium node Linux x86 cluster connected by Myrinet interconnect. However, Myrinet only connects nodes 1 through 15; as a result, the test cases show only up to 15 threads.

Michigan Tech's UPC (MuPC) is a runtime system (RTS) for UPC built on top of MPI[3]. It is portable to all the platforms that support MPI 1.1 and pthreads.

Berkeley's UPC (BUPC) is another RTS for UPC built on GASNet infrastructure which is supported over a wide variety of high-performance networks. GASNet (Global-Address-Space Networking) is a language-independent, low-level networking library developed at University of California, Berkeley.

[2] Eventually, the programmer has to synchronize all threads by calling a *barrier*.

[3] Message Passing Interface.

Both of the aforementioned runtime systems were used in testing the standard and extended collective reference implementations. A single test program was used to measure the performance of both forms of the collective functions to avoid any discrepancy due to testing. The program has been carefully designed following other past attempts in which after performing every collective operation, each thread does some local computation to reflect more realistic scenario. Further, the initialization overhead time is deducted from each measured completion time of the collective functions. Finally, the time that is reported reflects the average of 500 runs of each collective function on each platform. The source code for the testbed is in Appendix C.

6.2 Results

Each data point in the following graphs is an average over the maximum time taken by all threads in 500 runs. In other words, for every run the thread with the longest time is added to a total, and then divided by 500 to get an average of the maximums.

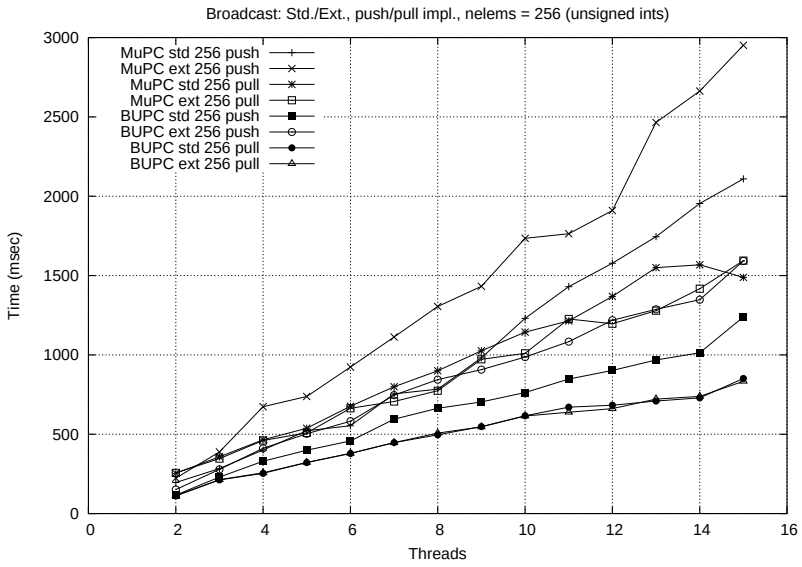


Figure 18: Broadcast: Standard and Extended implementations.

Figure 18 and 19 shows performance of the broadcast and exchange operations, both standard and extended versions, on MuPC and Berkeley UPC. The graphs demonstrate that the reference implementation of the extended collectives do not perform as well as the corresponding standard ones. However, this is an expected outcome, because the extra level of pointer redirection involved in the extended collective operations require extra address computation.

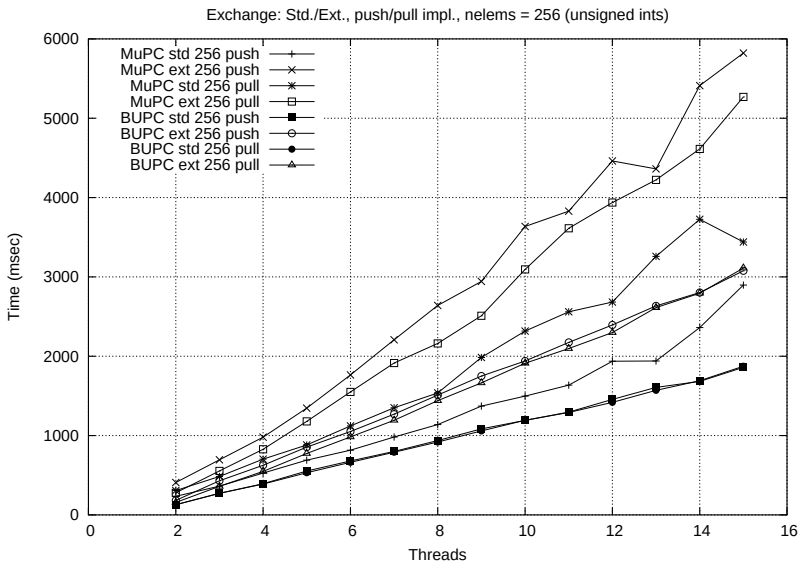


Figure 19: Exchange: Standard and Extended implementations.

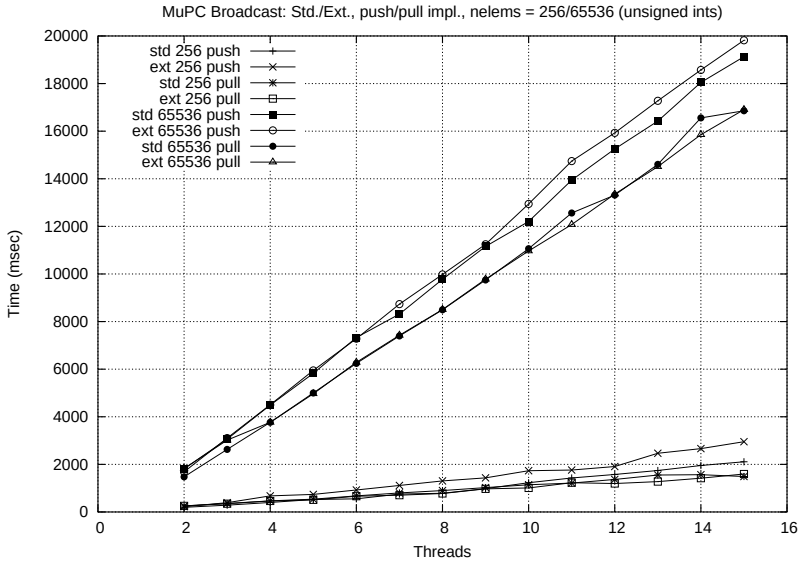


Figure 20: Effects of utilizing “push” and “pull” implementations of broadcast.

As the number of threads grow, both the implementations experience more time to complete. This is because of the amount of synchronization and communication involved.[4]

Implementation hint can be passed to the extended collective functions to take advan-

[4] `UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC` is OR-ed with the `mode` arguments.

tage of architecture depended performance improvements. Figure 20 shows results of both "push" and "pull"-based implementations of standard and extended broadcast operations on MuPC. The number of elements relocalized is varied between 256 and 65536. Each thread "pull"-ing data from the source thread performs better compared to the source thread "push"-ing data to all threads, one at a time. Moreover, the size of data block affects performance. Appendix A contains results of other collective operations.

7 Future Work

The reference implementation of the asynchronous collective relocalization operations are implemented as blocking. As a result, the completion functions return immediately. Berkeley's UPC provides non-blocking and non-contiguous memcpy functions, which can be used for a user-level implementation of the asynchronous collective operations. However, with MuPC further investigation is required to implement non-blocking versions of the extended collectives within MuPC.

Subsetting features have been specified, but not implemented. This is because of the complexity involved in providing an elegant solution. The obvious UPC way of grouping threads is by using the affinity factor. For example, the `upc_foreach` loop is a collective operation, where the fourth argument is checked to figure out which threads will execute the loop body.

8 Summary

The success of any programming language depends on its programmability and performance. UPC is a comparatively new and expressive parallel programming language. Although the programming model allows for remote read/write using simple assignment operations, a set of collective functions are provided to take advantage of the fact that bulk transfer of data is more efficient than individual transfers. To overcome the limitations of the standard collective functions and for better programmability factor a set of extended collective operations and one-sided collective operations are proposed in this work. Since the general versions of the extended collectives can handle the vector variants, the collective subcommittee has decided to drop the vector variants from the specification. As a result, the specification document only shows the general versions. Our performance results indicate acceptable reasons to adopt the extended collective specification to handle complex cases elegantly.

References

[1] Berkeley Unified Parallel C.
`http://upc.nersc.gov/download/dist/docs/html/`.

[2] GCC Unified Parallel C (GCC UPC).
`http://www.intrepid.com/upc/index.html`.

[3] HP Unified Parallel C (HP UPC).
`http://h30097.www3.hp.com/upc/`.

[4] MuPC: UPC Translator and Run Time Sytem.
`http://upc.mtu.edu/MuPCdistribution/index.html`.

[5] Co-array Fortran Performance and Potential: An NPB Experimental Study. In *Languages and Compilers for Parallel Computing (LCPC03)*, pages 177–193, College Station, TX, USA, 2003.

[6] William W. Carlson and Jesse M. Draper. Distributed data access in ac. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–47, New York, NY, USA, 1995. ACM Press.

[7] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in split-c. In *SC*, pages 262–273, 1993.

[8] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[9] E.D Brooks III, B.C. Gorda, and K.H. Warren. The Parallel C Preprocessor. In *Scientific Programming, vol. 1, no.1, pp 79-89*, 1992.

[10] P. Merkey and J. Purushothaman. Significant Precursors to UPC. Technical Report CS-TR-04-03, Michigan Technological University, July 2004.

[11] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[12] UPC Language Specifications V1.2. [online], June 2005.
`http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf`.

[13] E. Wiebel, D. Greenberg, and S. Seidel. UPC Collective Operations Specification V1.0, December 12 2003.
`http://www.gwu.edu/~upc/docs/UPC_Coll_Spec_V1.0.pdf`.

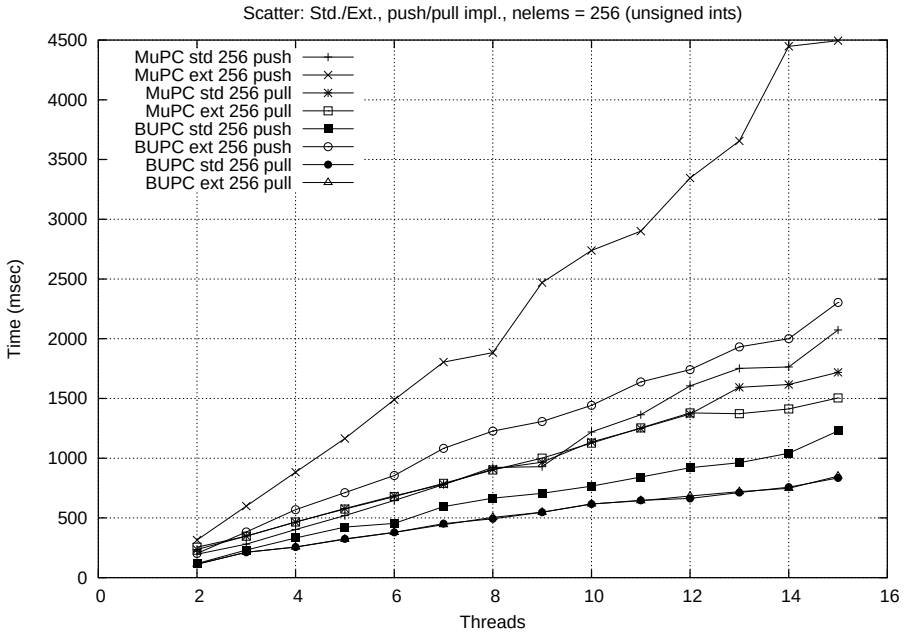
[14] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Concurrency: Practice and Experience, Vol. 10, No. 11-13*, September-November 1998. `http://titanium.cs.berkeley.edu/papers/titanium-hpj.ps`.

[15] Z. Ryne and S. Seidel. Ideas and Specification for the new One-sided Collective Operations in UPC, November 2005.

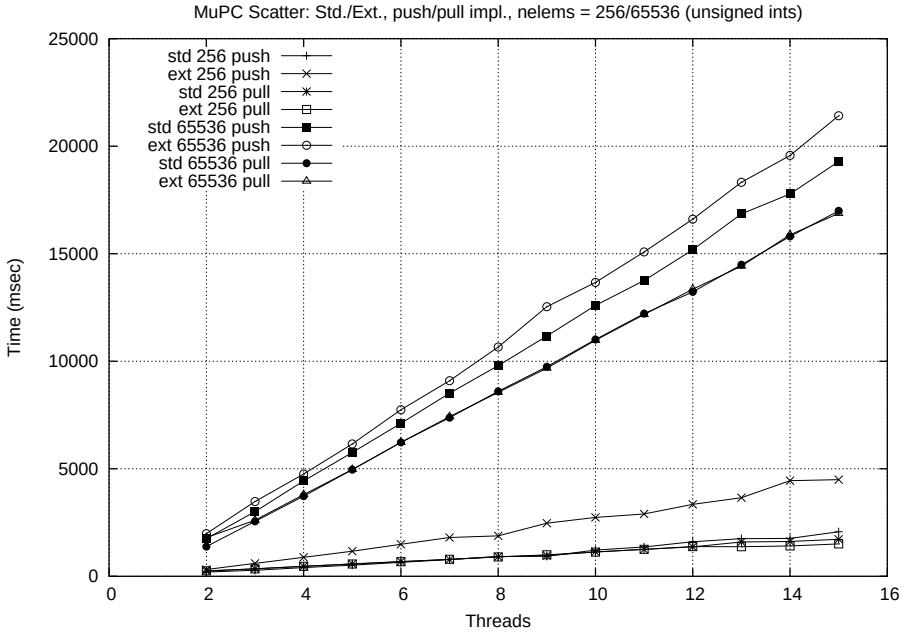
[16] Z. Ryne and S. Seidel. UPC Extended Collective Operations Specification (Draft), August 2005.

APPENDIX

A Performance Graphs

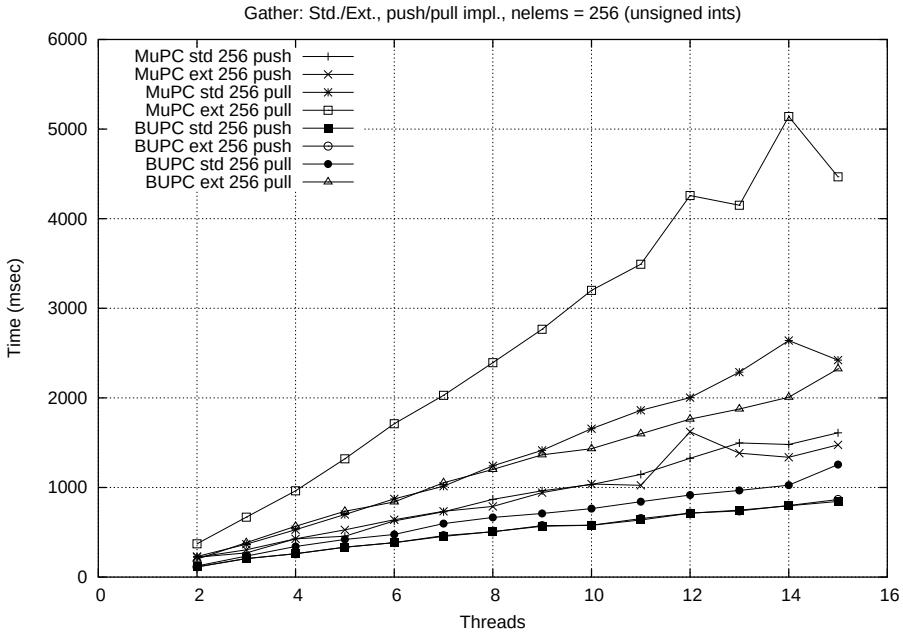


(a) Scatter: Standard and Extended implementations

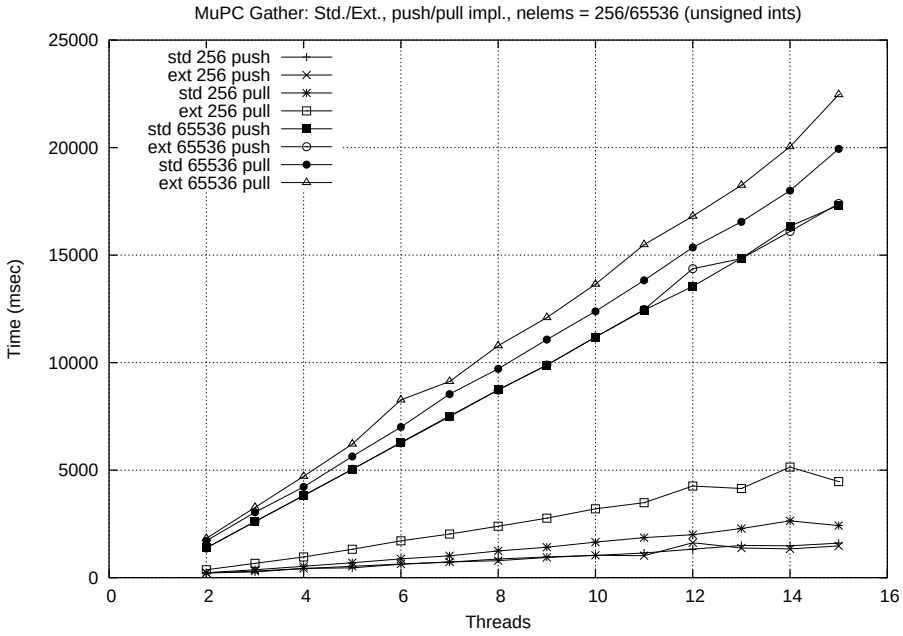


(b) Scatter: Vary `nelems`

Figure 21: Performance of Scatter: Standard vs Extended, with varying `nelems`

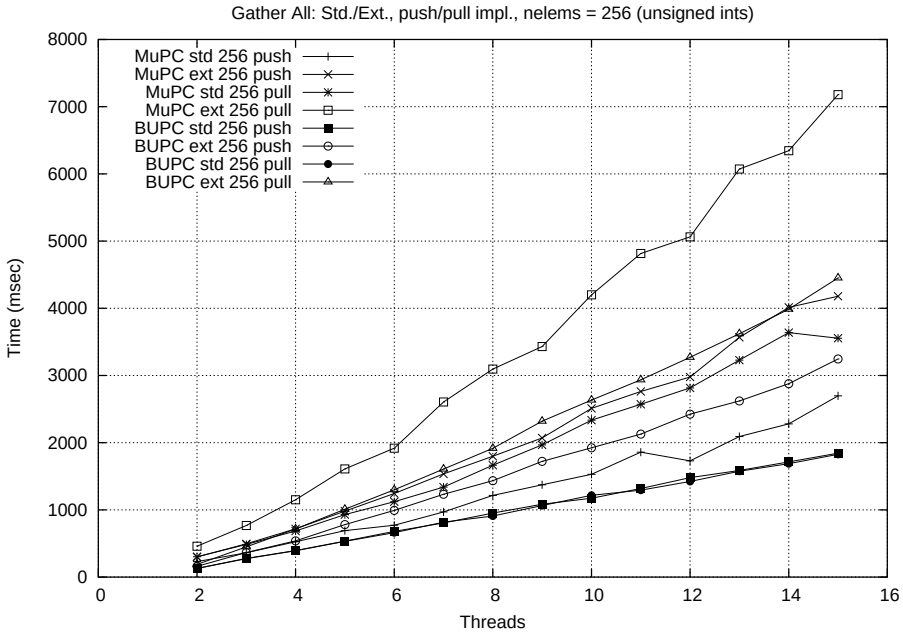


(a) Gather: Standard and Extended implementations

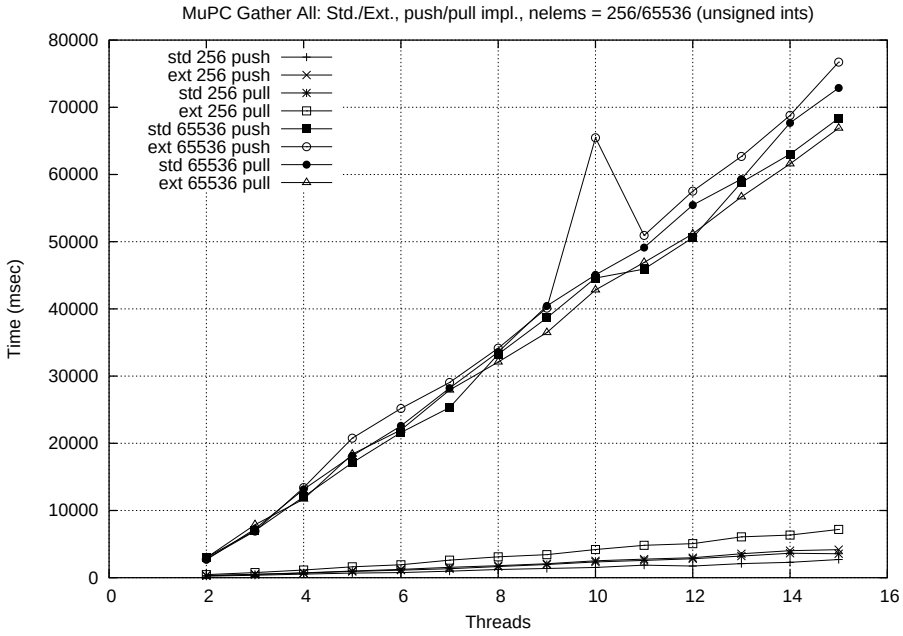


(b) Gather: Vary `nelems`

Figure 22: Performance of Gather: Standard vs Extended, with varying `nelems`

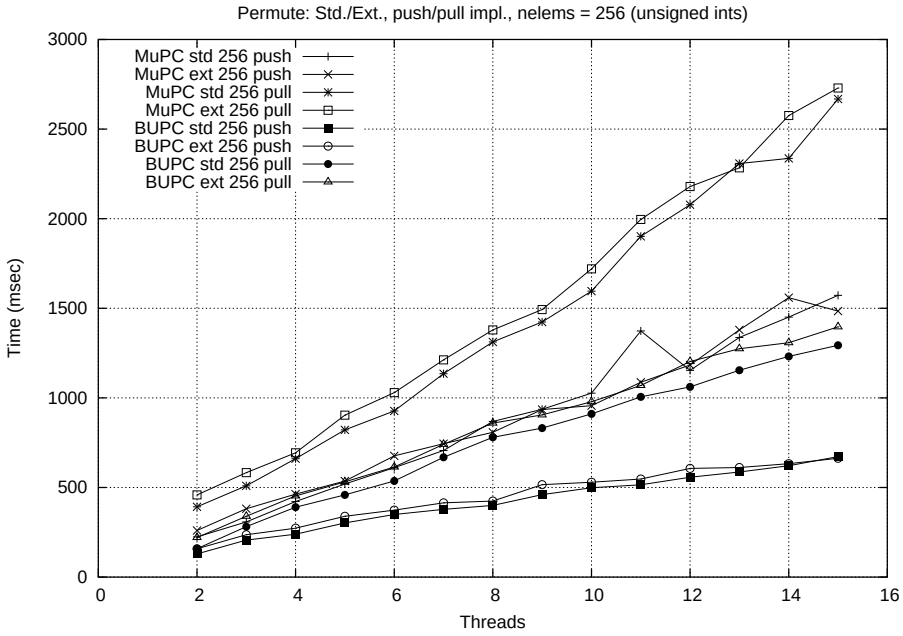


(a) Gather All: Standard and Extended implementations

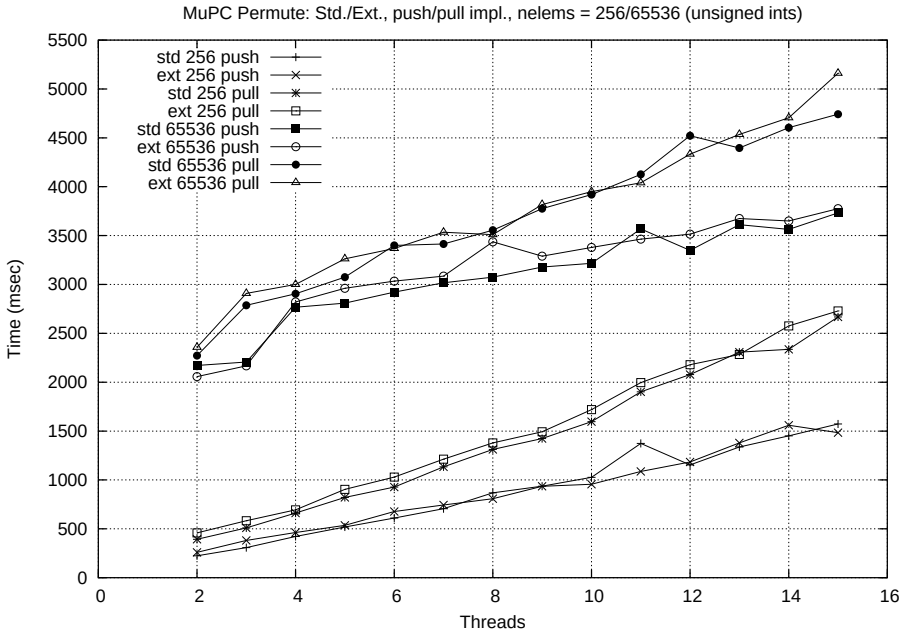


(b) Gather All: Vary `nelems`

Figure 23: Performance of Gather All: Standard vs Extended, with varying `nelems`



(a) Permute: Standard and Extended implementations



(b) Permute: Vary `nelems`

Figure 24: Performance of Permute: Standard vs Extended, with varying `nelems`

B Collective Operation Implementations

B.1 Broadcast - Standard version: Implementation

```
/*****************************************************************************/
/*                                                                           */
/*        UPC collective function library, reference implementation          */
/*                                                                           */
/*   Steve Seidel, Dept. of Computer Science, Michigan Technological Univ.   */
/*   steve@mtu.edu                                      February 6, 2004   */
/*                                                                           */
/*****************************************************************************/

void upc_all_broadcast( shared void *dst,
shared const void *src,
size_t nbytes,
upc_flag_t sync_mode )
{
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
// Synchronize using barriers in the cases of MYSYNC and ALLSYNC.
if ( UPC_IN_MYSYNC & sync_mode || !(UPC_IN_NOSYNC & sync_mode) )
upc_barrier;

#ifdef PULL
// Each thread "pulls" the data from the source thread.
upc_memcpy( (shared char *)dst + MYTHREAD, (shared char *)src, nbytes );
#endif

#ifdef PUSH
int i;

// The source thread "pushes" the data to each destination.
if ( upc_threadof((shared void *)src) == MYTHREAD )
{
for (i=0; i<THREADS; ++i)
{
upc_memcpy( (shared char *)dst + i,
(shared char *)src, nbytes );
}
}
#endif

// Synchronize using barriers in the cases of MYSYNC and ALLSYNC.
if ( UPC_OUT_MYSYNC & sync_mode || !(UPC_OUT_NOSYNC & sync_mode) )
upc_barrier;
}
```

B.2 Broadcast - Vector variant: Implementation

```
/*****************************************************************************/
/*                                                                           */
/*    UPC extended collective function library, reference implementation     */
/*                                                                           */
/*  Zinnu Ryne, Dept. of Computer Science, Michigan Technological University */
/*  zryne@mtu.edu                                                 May, 2005 */
/*                                                                           */
/*****************************************************************************/

#include <upc.h>
#include "upc_collective_ext.h"

//---------------------------------------------------------------------------
// Broadcast: Vector Variant
//---------------------------------------------------------------------------
void upc_all_broadcast_v( shared void * dst,
                          shared const void * src,
                          shared size_t * ddisp,/* array indexes */
                          size_t nelems,/* number of elements */
                          size_t dst_blk,/* blocking factor */
  size_t typesize,/* typesize in bytes */
                          upc_flag_t mode )
{
   int i;

   //---------------------------------------------------------------------------
   // Synchronize using barriers in the cases of MYSYNC and ALLSYNC.
   //---------------------------------------------------------------------------
   if( (UPC_IN_MYSYNC & mode) || !(UPC_IN_NOSYNC & mode) )
   {
      upc_barrier;
   }

   //---------------------------------------------------------------------------
   // PULL implementation (default):
   // Each thread "pulls" the data from the source thread.
   //---------------------------------------------------------------------------
   if( ((UPC_PULL & mode) && !(UPC_PUSH & mode)) || // only PULL specified
       (!(UPC_PULL & mode) && !(UPC_PUSH & mode)) ) // or, nothing.
   {
      //---------------------------------------------------------------------------
      // ADDRESS COMPUTATION:
      //
      // The source and number of elements to broadcast arguments are same for
      // all threads; destination is the only thing that varies depending on
      // thread. Once the 'shared void *' typecasting is done, we loose the
      // blocking information; that is why we have to figure out the exact
      // mapping.
      //
      // The destination argument to 'upc_memcpy' is a optimized one from a
      // more complex variant. Here we try to explain how we do the mapping,
      // with a formal proof for the optimization.
      //
      // We start with the following destination argument to pass to upc_memcpy:
      //
      //    (shared char *)dst
      // + (dst_blk*typesize)
      //   * ((ddisp[MYTHREAD]*typesize)/(dst_blk*typesize)
      //     - (((ddisp[MYTHREAD]*typesize)/(dst_blk*typesize))%THREADS))
```

```
// + ((ddisp[MYTHREAD]*typesize)/(dst_blk*typesize))%THREADS
// + ((ddisp[MYTHREAD]*typesize)%(dst_blk*typesize))*THREADS
//
// For reference:
//  t = THREADS
//  d = disp[MYTHREAD]*typesize
//  b = dst_blk*typesize
//
// So the computation is:
// b*((d/b) - ((d/b)%t)) + (d/b)%t + (d%b)*t
//
// Explanation:
// [1] b*((d/b)-((d/b)%t)) gives us the index of the first element of a
//     block on thread 0. That block is in the same relative location as
//     the block on MYTHREAD that contains the element at displacement d.
// [2] (d/b)%t is the thread number that contains the desired block, i.e.,
//     MYTHREAD.
// [3] (d%b)t is the phase of the desired element within its block.
//
// We can optimize the first part using the claim that
// b*((d/b) - ((d/b)%t)) = (d/(tb))tb
//
// -------------------------------------
// | A FORMAL PROOF (by Steve Seidel) |
// -------------------------------------
//
// Claim: b(d/b - (d/b) % t) = (d/(tb))tb
//
// Proof: The last b's cancel, so it is sufficient to prove that
//  d/b - (d/b) % t = (d/(tb))t.
//
// For some j and k such that 0 <= j < tb and k >= 0,
// it is true that
//  d = ktb+j.
//
// Then for the right side of this claim,
// (d/(tb))t = ( (ktb+j)/(tb) )t
//      = ( k + j/(tb) )t
//      = kt
//
// Since j < tb, so j/(tb) = 0.
//
// And for the left side,
//  d/b - (d/b) % t = (ktb+j)/b - ((ktb+j)/b) % t
// = kt + j/b - (kt + j/b) % t
// = kt + j/b - (j/b) % t
// = kt
//
// Since j/b < t, so (j/b) % t = j/b.
//
// So using this claim, we can have the following:
//
//    (shared char *)dst
// + ((ddisp[MYTHREAD]*typesize)/(THREADS*(dst_blk*typesize)))
// *THREADS*(dst_blk*typesize)
// + ((ddisp[MYTHREAD]*typesize)/(dst_blk*typesize))%THREADS
// + ((ddisp[MYTHREAD]*typesize)%(dst_blk*typesize))*THREADS
//
// And finally, the following:
//
//    (shared char *)dst
```

```
      // + (ddisp[MYTHREAD]/(THREADS*dst_blk))*THREADS*dst_blk*typesize
      // + (ddisp[MYTHREAD]/dst_blk)%THREADS
      // + ((ddisp[MYTHREAD]*typesize)%(dst_blk*typesize))*THREADS,
      //
      //-------------------------------------------------------------------------

      //-------------------------------------------------------------------------
      // If IN_PLACE is specified, move data for all but the source thread
      //-------------------------------------------------------------------------
      if( (UPC_IN_PLACE & mode) )
      {
 if( upc_threadof( (shared void *)src ) != MYTHREAD )
 {
    upc_memcpy( (shared char *)dst
  + (ddisp[MYTHREAD]/(THREADS*dst_blk))*THREADS*dst_blk*typesize
  + (ddisp[MYTHREAD]/dst_blk)%THREADS
  + ((ddisp[MYTHREAD]*typesize)%(dst_blk*typesize))*THREADS,
(shared const char *)src,
nelems*typesize );
 }
      }
      //-------------------------------------------------------------------------
      // Otherwise, move data for all threads
      //-------------------------------------------------------------------------
      else
      {
 upc_memcpy( (shared char *)dst
  + (ddisp[MYTHREAD]/(THREADS*dst_blk))*THREADS*dst_blk*typesize
  + (ddisp[MYTHREAD]/dst_blk)%THREADS
  + ((ddisp[MYTHREAD]*typesize)%(dst_blk*typesize))*THREADS,
       (shared const char *)src,
       nelems*typesize );
      }
   }
   //-------------------------------------------------------------------------
   // PUSH implementation:
   // The source thread "pushes" the data to each destination.
   //-------------------------------------------------------------------------
   else if( (UPC_PUSH & mode) && !(UPC_PULL & mode) ) // only PUSH specified
   {
      //-------------------------------------------------------------------------
      // Find the source thread
      //-------------------------------------------------------------------------
      if( upc_threadof( (shared void *)src ) == MYTHREAD )
      {
 //-----------------------------------------------------------------------
 // If IN_PLACE is specified, move data for all but the source thread
 //-----------------------------------------------------------------------
 if( (UPC_IN_PLACE & mode) )
 {
    //-------------------------------------------------------------------
    // Push each block of data to the destination threads, one at a time
    //-------------------------------------------------------------------
    for( i = 0; i < THREADS; i++ )
    {
       if( upc_threadof( (shared void *)src ) != i )
       {
  upc_memcpy( (shared char *)dst
      + (ddisp[i]/(THREADS*dst_blk))*THREADS*dst_blk*typesize
      + (ddisp[i]/dst_blk)%THREADS
      + ((ddisp[i]*typesize)%(dst_blk*typesize))*THREADS,
```

```
    (shared const char *)src,
    nelems*typesize );
       }
    }
 }
 //------------------------------------------------------------------------
 // Otherwise, move data for all threads
 //------------------------------------------------------------------------
 else
 {
    //---------------------------------------------------------------------
    // Push each block of data to the destination threads, one at a time
    //---------------------------------------------------------------------
    for( i = 0; i < THREADS; i++ )
    {
       upc_memcpy( (shared char *)dst
   + (ddisp[i]/(THREADS*dst_blk))*THREADS*dst_blk*typesize
   + (ddisp[i]/dst_blk)%THREADS
   + ((ddisp[i]*typesize)%(dst_blk*typesize))*THREADS,
(shared const char *)src,
nelems*typesize );
    }
 }
       }
   }
   //-------------------------------------------------------------------------
   // The user provided both UPC_PUSH and UPC_PULL in 'mode'.
   //-------------------------------------------------------------------------
   else
   {
      // This is an error case which must be caught and reported!
   }

   //-------------------------------------------------------------------------
   // Synchronize using barriers in the cases of MYSYNC and ALLSYNC.
   //-------------------------------------------------------------------------
   if( (UPC_OUT_MYSYNC & mode) || !(UPC_OUT_NOSYNC & mode) )
   {
      upc_barrier;
   }
}
```

B.3 Broadcast - Generalized version: Implementation

```
/*****************************************************************************/
/*                                                                           */
/*    UPC extended collective function library, reference implementation     */
/*                                                                           */
/*  Zinnu Ryne, Dept. of Computer Science, Michigan Technological University */
/*  zryne@mtu.edu                                          September, 2005 */
/*                                                                           */
/*****************************************************************************/

#include <upc.h>
#include "upc_collective_ext.h"

//---------------------------------------------------------------------------
// Extended Broadcast
//---------------------------------------------------------------------------
void upc_all_broadcast_x( shared void * shared * dst, shared const void * src,
                          size_t nbytes, upc_flag_t mode )
{
   int i;

   //---------------------------------------------------------------------------
   // Synchronize using barriers in the cases of MYSYNC and ALLSYNC.
   //---------------------------------------------------------------------------
   if( (UPC_IN_MYSYNC & mode) || !(UPC_IN_NOSYNC & mode) )
   {
      upc_barrier;
   }

   //---------------------------------------------------------------------------
   // PULL implementation (default):
   // Each thread "pulls" the data from the source thread.
   //---------------------------------------------------------------------------
   if( ((UPC_PULL & mode) && !(UPC_PUSH & mode)) || // only PULL specified
       (!(UPC_PULL & mode) && !(UPC_PUSH & mode)) ) // or, nothing
   {
      //---------------------------------------------------------------------------
      // If IN_PLACE is specified, move data for all but the source thread
      //---------------------------------------------------------------------------
      if( (UPC_IN_PLACE & mode) )
      {
 if( upc_threadof( (shared void *)src ) != MYTHREAD )
 {
    upc_memcpy( dst[MYTHREAD], src, nbytes );
 }
      }
      //---------------------------------------------------------------------------
      // Otherwise, move data for all threads
      //---------------------------------------------------------------------------
      else
      {
 upc_memcpy( dst[MYTHREAD], src, nbytes );
      }
   }
   //---------------------------------------------------------------------------
   // PUSH implementation:
   // The source thread "pushes" the data to each destination.
   //---------------------------------------------------------------------------
   else if( (UPC_PUSH & mode) && !(UPC_PULL & mode) ) // only PUSH specified
   {
```

```
      //------------------------------------------------------------------------
      // Find the source thread.
      //------------------------------------------------------------------------
      if( upc_threadof( (shared void *)src ) == MYTHREAD )
      {
 //---------------------------------------------------------------------
 // If IN_PLACE is specified, move data for all but the source thread
 //---------------------------------------------------------------------
 if( (UPC_IN_PLACE & mode) )
 {
    //---------------------------------------------------------------
    // Push each block of data to the destination threads, one at a time
    //---------------------------------------------------------------
    for( i = 0; i < THREADS; i++ )
    {
       if( upc_threadof( (shared void *)src ) != i )
       {
  upc_memcpy( dst[i], src, nbytes );
       }
    }
 }
 //---------------------------------------------------------------------
 // Otherwise, move data for all threads
 //---------------------------------------------------------------------
 else
 {
    //---------------------------------------------------------------
    // Push each block of data to the destination threads, one at a time
    //---------------------------------------------------------------
    for( i = 0; i < THREADS; i++ )
    {
       upc_memcpy( dst[i], src, nbytes );
    }
 }
      }
   }
   //------------------------------------------------------------------------
   // The user provided both UPC_PUSH and UPC_PULL in 'mode'.
   //------------------------------------------------------------------------
   else
   {
      // This is an error case which must be caught and reported!
      upc_coll_warning( 1 );
   }

   //------------------------------------------------------------------------
   // Synchronize using barriers in the cases of MYSYNC and ALLSYNC.
   //------------------------------------------------------------------------
   if( (UPC_OUT_MYSYNC & mode) || !(UPC_OUT_NOSYNC & mode) )
   {
      upc_barrier;
   }
}
```

C Testbed Source

C.1 Setup and Execution Script

```
#!/bin/bash

##############################################################################
#
# This script is used for performance results only!
# It compiles the sources with appropriate flags, runs the executables,
# collects data in a file, and then runs GNUPlot to create EPS files, all stored
# in $(PRES) directory. Finally it converts all the EPS files to PDFs.
#
#
# Date   : November 25, 2005
# Author : Zinnu Ryne
# Email  : zryne@mtu.edu
#
##############################################################################

PDIR="performance"
PRES="$PDIR/results"

MUPC="/home/zhazhang/MuPC-working/bin/mupcc"
BUPC="/home/zhazhang/Berkeley/bin/upcc"

make clean

##############################################################################
# Relocate 'n' elements or 'n*sizeof(TYPE)' bytes
##############################################################################
for n in 256 65536
do
   ###########################################################################
   # Apply implementation 'm'
   ###########################################################################
   for m in PUSH PULL
   do
      ########################################################################
      # Test on runtime system 'k'
      ########################################################################
      for k in $MUPC $BUPC
      do
 if [ "$k" = "$MUPC" ];
 then
    rts="mpirun"
    thdflag="-f"
    optflag="-D$m"
 elif [ "$k" = "$BUPC" ];
 then
    rts="/home/zhazhang/Berkeley/bin/upcrun -q"
    thdflag="-T"
    #optflag="-network=gm -D$m"
    optflag="-norc -network=mpi -D$m"
 else
    echo "ERROR: Runtime System not defined!"
 fi

         ###################################################################
 # Compile/run relocalization collective 'j'
```

```
        ##########################################################################
 for j in pbcast pbcastx pscat pscatx pgath pgathx pgall pgallx pexch pexchx pperm ppermx
 do
    if [ "$k" = "$MUPC" ];
    then
       echo "# $j on MuPC, $m $n elements" > "$PRES/$j-$m-$n-average-MUPC.txt"
    elif [ "$k" = "$BUPC" ];
    then
       echo "# $j on BUPC, $m $n elements" > "$PRES/$j-$m-$n-average-BUPC.txt"
    fi

            ##########################################################################
    # Compile/run results for 'i' threads
            ##########################################################################
    for i in 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    do
       echo ""
       echo " -- make COM=$k"
       echo " ------- OPF=$optflag NUM=$n TFLG=$thdflag THD=$i $j"
       make COM="$k" OPF="$optflag" NUM="$n" TFLG="$thdflag" THD="$i" "$j"

       if [ "$k" = "$MUPC" ];
       then
  make RUN="$rts" TFLG="$thdflag" THD="$i" "$j-run" >> "$PRES/$j-$m-$n-average-MUPC.txt"
       elif [ "$k" = "$BUPC" ];
       then
  make RUN="$rts" TFLG="$thdflag" THD="$i" "$j-run" >> "$PRES/$j-$m-$n-average-BUPC.txt"
       fi

       make clean
       sleep 5
    done
 done
      done
   done
done
```

C.2 Test Program

```
//--------------------------------------------------------------------------
//
// Author: Zinnu Ryne
// Email: zryne@mtu.edu
//
// This program is intended for a comparative performance analysis of the
// extended collectives with regards to the standard ones. For simplicity,
// all code for performance analysis is put in one source file.
//
//
// Start date: 11/09/2005
// Finalized : 12/13/2005
//
// COMPILE: mupcc -D_<STD|EXT>_COLL
//                -D_<STD|EXT>_<BCAST|SCAT|GATH|GALL|EXCH|PERM>
//   [-DNELEMS=1024]
//                -f <threads> -o <exec> upc_all_... Performance.c
//
// DISCLAIMER: This code is based on Jaisudha Purushothaman's performance
//             testsuite for her Prefix-Reduce implementation.
//
//--------------------------------------------------------------------------

#include <upc.h>
#include <time.h>
#include <sys/time.h>

#ifdef _STD_COLL
#include "../src/std/upc_collective.h"
#endif

#ifdef _EXT_COLL
#include "../src/ext/upc_collective_ext.h"
#endif

//--------------------------------------------------------------------------
// All defined macros
//--------------------------------------------------------------------------
#define TYPE unsigned int // The data-type we're using

#ifndef NELEMS
#define NELEMS 4096 // Number of elements to relocate
#endif

#define SRC_BLK_SIZE NELEMS*THREADS // Block size of the source array
#define DST_BLK_SIZE NELEMS*THREADS // Block size of the destination array
#define SRC_SIZE SRC_BLK_SIZE*THREADS // Size of the source array
#define DST_SIZE DST_BLK_SIZE*THREADS // Size of the destination array
#define TIME_ITERATIONS 100   // Iterations in overhead timing
#define TEST_ITERATIONS 500 // Iterations in real testing
#define COMP_ITERATIONS 50

//--------------------------------------------------------------------------
// Data storage
//--------------------------------------------------------------------------
shared [SRC_BLK_SIZE] TYPE src[SRC_SIZE];
shared [DST_BLK_SIZE] TYPE dst[DST_SIZE];
shared [1] size_t perm[THREADS];
```

```
#ifndef _EXT_EXCH
shared [] TYPE * shared [1] mySrc[THREADS];
#ifndef _EXT_GALL
shared [] TYPE * shared [1] myDst[THREADS];
#endif
#ifdef _EXT_GALL
shared [] TYPE * shared [THREADS] myDst[THREADS*THREADS];
#endif
shared [1] size_t nBytes[THREADS];
#endif

#ifdef _EXT_EXCH
shared [] TYPE * shared [THREADS] mySrc[THREADS*THREADS];
shared [] TYPE * shared [THREADS] myDst[THREADS*THREADS];
shared [THREADS] size_t nBytes[THREADS*THREADS];
#endif

//--------------------------------------------------------------------------
// Timing storage
//--------------------------------------------------------------------------
shared [1] unsigned int measured[THREADS];
shared double avgtime, max, mintime;

//--------------------------------------------------------------------------
// Function to calculate time differeces.
// Subtract the ‘struct timeval’ values X and Y, and store the result in RESULT.
// Return 1 if the difference is negative, otherwise 0.
//--------------------------------------------------------------------------
int timeval_subtract (result, x, y) struct timeval *result, *x, *y;
{
   // Perform the carry for the later subtraction by updating y.
   if( x->tv_usec < y->tv_usec )
   {
      int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
      y->tv_usec -= 1000000 * nsec;
      y->tv_sec += nsec;
   }

   if( x->tv_usec - y->tv_usec > 1000000 )
   {
      int nsec = (y->tv_usec - x->tv_usec) / 1000000;
      y->tv_usec += 1000000 * nsec;
      y->tv_sec -= nsec;
   }

   // Compute the time remaining to wait. tv_usec is certainly positive.
   result->tv_sec = x->tv_sec - y->tv_sec;
   result->tv_usec = x->tv_usec - y->tv_usec;

   // Return 1 if result is negative.
   return x->tv_sec < y->tv_sec;
}

//--------------------------------------------------------------------------
// MAIN
//--------------------------------------------------------------------------
int main()
{
   int i, j;
   int temp, l, m, n;
   struct timeval time_start, time_stop, time_diff;
```

```
   double overhead = 0.0;
   size_t dataSize;

   //---------------------------------------------------------------------
   // STEP 1: Calculate the average overhead of getting time.
   //---------------------------------------------------------------------
   for( i = 0; i < TIME_ITERATIONS; i++ )
   {
      gettimeofday( &time_start, NULL );
      gettimeofday( &time_stop, NULL );

      if( timeval_subtract( &time_diff, &time_stop, &time_start ) )
      {
 printf( "ERROR: Th[%d] has negative time, so exiting!\n", MYTHREAD );
 exit( 1 );
      }

      //*********************************************************************
      // Total overhead
      //*********************************************************************
      overhead += time_diff.tv_sec * 1000000.0 + time_diff.tv_usec;
   }

   //*********************************************************************
   // Average overhead:
   // Uncomment the print function to see the average overhead per thread.
   //*********************************************************************
   overhead /= TIME_ITERATIONS;
   //printf( "Th[%d]: Average overhead =  %f\n", MYTHREAD, overhead );

   upc_barrier;

   //---------------------------------------------------------------------
   // STEP 2: Initialize all necessary data.
   //---------------------------------------------------------------------
   dataSize = NELEMS*sizeof(TYPE); // Used for standard collectives,
        // instead of nBytes. This is local to
// all threads.
#ifndef _EXT_EXCH
   nBytes[MYTHREAD] = NELEMS*sizeof(TYPE);
#endif
#ifdef _EXT_EXCH
   for( i = 0; i < THREADS; i++ )
   {
      nBytes[MYTHREAD*THREADS+i] = NELEMS*sizeof(TYPE);
   }
#endif

   perm[MYTHREAD] = THREADS - MYTHREAD - 1;

   if( MYTHREAD == 0 )
   {
      for( i = 0; i < SRC_SIZE; i++ )
      {
 src[i] = (char)(1 + i);
      }

      for( i = 0; i < THREADS; i++ )
      {
#ifdef _EXT_BCAST
 myDst[i] = (shared [] TYPE *)&(dst[i*DST_BLK_SIZE]);
```

```
#endif
#ifdef _EXT_SCAT
 mySrc[i] = (shared [] TYPE *)&(src[i*NELEMS]);
 myDst[i] = (shared [] TYPE *)&(dst[i*DST_BLK_SIZE]);
#endif
#ifdef _EXT_GATH
 mySrc[i] = (shared [] TYPE *)&(src[i*SRC_BLK_SIZE]);
 myDst[i] = (shared [] TYPE *)&(dst[i*NELEMS]);
#endif
#ifdef _EXT_GALL
 mySrc[i] = (shared [] TYPE *)&(src[i*SRC_BLK_SIZE]);
 for( j = 0; j < THREADS; j++ )
 {
    myDst[i*THREADS+j] = (shared [] TYPE *)&(dst[(i*THREADS+j)*NELEMS]);
 }
#endif
#ifdef _EXT_EXCH
 for( j = 0; j < THREADS; j++ )
 {
    mySrc[i*THREADS+j] = (shared [] TYPE *)&(src[(i*THREADS+j)*NELEMS]);
    myDst[i*THREADS+j] = (shared [] TYPE *)&(dst[(i*THREADS+j)*NELEMS]);
 }
#endif
#ifdef _EXT_PERM
 mySrc[i] = (shared [] TYPE *)&(src[i*SRC_BLK_SIZE]);
 myDst[i] = (shared [] TYPE *)&(dst[i*DST_BLK_SIZE]);
#endif
      }
   }

   upc_barrier;

   //------------------------------------------------------------------------
   // STEP 3: Run the actual test.
   //        NOTE that we run 1 extra test to discard the first test case.
   //------------------------------------------------------------------------
   for( j = 0; j <= TEST_ITERATIONS; j++ )
   {
      gettimeofday( &time_start, NULL );

#ifdef _STD_BCAST
      upc_all_broadcast( dst, src, dataSize, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
#endif

#ifdef _EXT_BCAST
#ifdef PULL
      upc_all_broadcast_x( myDst, src, dataSize,
   UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
#endif
#ifdef PUSH
      upc_all_broadcast_x( myDst, src, dataSize,
   UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH );
#endif
#endif

#ifdef _STD_SCAT
      upc_all_scatter( dst, src, dataSize, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
#endif

#ifdef _EXT_SCAT
#ifdef PULL
```

```
      upc_all_scatter_x( myDst, mySrc, nBytes,
 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
#endif
#ifdef PUSH
      upc_all_scatter_x( myDst, mySrc, nBytes,
 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH );
#endif
#endif

#ifdef _STD_GATH
      upc_all_gather( dst, src, dataSize, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
#endif

#ifdef _EXT_GATH
#ifdef PULL
      upc_all_gather_x( myDst, mySrc, nBytes,
UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
#endif
#ifdef PUSH
      upc_all_gather_x( myDst, mySrc, nBytes,
UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH );
#endif
#endif

#ifdef _STD_GALL
      upc_all_gather_all( dst, src, dataSize, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
#endif

#ifdef _EXT_GALL
#ifdef PULL
      upc_all_gather_all_x( myDst, mySrc, nBytes,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
#endif
#ifdef PUSH
      upc_all_gather_all_x( myDst, mySrc, nBytes,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH );
#endif
#endif

#ifdef _STD_EXCH
      upc_all_exchange( dst, src, dataSize, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
#endif

#ifdef _EXT_EXCH
#ifdef PULL
      upc_all_exchange_x( myDst, mySrc, nBytes,
  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
#endif
#ifdef PUSH
      upc_all_exchange_x( myDst, mySrc, nBytes,
  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH );
#endif
#endif

#ifdef _STD_PERM
      upc_all_permute( dst, src, perm, dataSize,
       UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
#endif

#ifdef _EXT_PERM
#ifdef PULL
```

```
      upc_all_permute_x( myDst, mySrc, perm, nBytes,
 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
#endif
#ifdef PUSH
      upc_all_permute_x( myDst, mySrc, perm, nBytes,
 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH );
#endif
#endif

      gettimeofday( &time_stop, NULL );

      //************************************************************************
      // Some computation to keep the thread busy. Basically, garbage!
      //************************************************************************
      for( l = 0; l < COMP_ITERATIONS; l++ )
      {
 for ( m = 0; m < COMP_ITERATIONS; m++ )
 {
    for( n = 0; n < COMP_ITERATIONS; n++ )
    {
       temp = l + m * n;
    }
 }
      }
      temp = temp + 1;

      //************************************************************************
      // Elapsed time in microseconds = 1E+06 seconds.
      //************************************************************************
      if( timeval_subtract( &time_diff, &time_stop, &time_start ) )
      {
 printf( "ERROR: Th[%d] has negative time, so exiting!\n", MYTHREAD );
 exit( 1 );
      }

      measured[MYTHREAD]= time_diff.tv_sec * 1000000.0
   + time_diff.tv_usec - overhead;

      upc_barrier;

      if( MYTHREAD == 0 )
      {
 max = 0.0;

 //********************************************************************
 // For each run, get the longest time from all threads
 //********************************************************************
 for( i = 0; i < THREADS; i++ )
 {
    if( measured[i] > max )
    {
       max = measured[i];
    }
 }

 //********************************************************************
 // Ignore the first run in calculating the sum of all max's.
 //********************************************************************
 if( j != 0 )
 {
    avgtime += max;
```

```
 }

 //*********************************************************************
 // Uncomment the following if acquiring all data
 // NOTE: (1) Comment out the average data printouts below.
 //       (2) Use the runtest.alldata to get results.
 //*********************************************************************
 //printf( "%f\n", max );

 if( j == 0 )
 {
    mintime = max;
 }
 else if( max < mintime )
 {
    mintime = max;
 }
      } // end of if
   } // end of for

   //---------------------------------------------------------------------------
   // STEP 4: A single thread will collect and report data.
   //---------------------------------------------------------------------------
   if( MYTHREAD == 0 )
   {
      avgtime /= TEST_ITERATIONS;
      //*********************************************************************
      // Uncomment the following if acquiring only average.
      // NOTE: (1) Comment out the individual data printouts above.
      //       (2) Use the runtest.average to get results.
      //*********************************************************************
      printf( "%d\t%10f\t%10f\n", THREADS, mintime, avgtime );
   }

   //*****************************************************************************
   // Just so that the compiler doesn't act smart!
   //*****************************************************************************
   dataSize = dataSize * 2;
}
```