

# UPC Extended Collective Operations Specification

*Version 0.3*

Zinnu Ryne, Steven R. Seidel  
{zryne, steve}@mtu.edu  
Department of Computer Science  
Michigan Technological University  
Houghton, MI 49931

With contributions from  
Paul H. Hargrove, Dan Bonachea, Rajesh Nishtala  
Berkeley UPC Group

December 15, 2005

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Scope</b>	<b>1</b>
<b>3</b>	<b>Definitions</b>	<b>1</b>
<b>4</b>	<b>Common Requirements</b>	<b>2</b>
<b>5</b>	<b>Extended Collectives Library</b>	<b>3</b>
5.1	Standard header . . . . .	3
5.2	Predefined Constants . . . . .	3
5.3	Extended Relocalization Operations . . . . .	3
5.3.1	The <code>upc_all_broadcast_x</code> function . . . . .	4
5.3.2	The <code>upc_all_scatter_x</code> function . . . . .	5
5.3.3	The <code>upc_all_gather_x</code> function . . . . .	7
5.3.4	The <code>upc_all_gather_all_x</code> function . . . . .	9
5.3.5	The <code>upc_all_exchange_x</code> function . . . . .	11
5.3.6	The <code>upc_all_permute_x</code> function . . . . .	13
5.4	Completion Operations . . . . .	16
5.4.1	The <code>upc_wait</code> function . . . . .	16
5.4.2	The <code>upc_test</code> function . . . . .	16
5.4.3	The <code>upc_waitany</code> function . . . . .	16
5.4.4	The <code>upc_testany</code> function . . . . .	17
5.4.5	The <code>upc_waitall</code> function . . . . .	17
5.4.6	The <code>upc_testall</code> function . . . . .	17
5.4.7	The <code>upc_waitsome</code> function . . . . .	18
5.4.8	The <code>upc_testsome</code> function . . . . .	18
5.4.9	The <code>upc_get_status</code> function . . . . .	18
5.5	Extended Computational Operations . . . . .	19
5.5.1	The <code>upc_all_reduce_all</code> function and Exclusive Prefix-reduction . . . . .	19
	<b>References</b>	<b>21</b>

## List of Figures

1	upc_all_broadcast_x . . . . .	4
2	upc_all_scatter_x . . . . .	6
3	upc_all_gather_x . . . . .	8
4	upc_all_gather_all_x . . . . .	10
5	upc_all_exchange_x . . . . .	12
6	upc_all_permute_x . . . . .	15

## List of Tables

# 1 Introduction

1. The earliest version of the UPC Collective Specifications [3] was authored by Elizabeth Wiebel and David Greenberg and appeared as CCS-TR-02-159 in March 2002. UPC collectives have been discussed at workshops at SC2001, SC2002, SC2003, and at workshops held at George Washington University in March 2002, May 2003, and September 2005.
2. The current version (V1.0) of the UPC Collective Specifications [4, 1], published in December 12, 2003, is now part of the UPC Language Specifications V1.2 [2].
3. The need for extended collective operations was discussed in the UPC collectives subgroup and at SC2004. Further refining was done at the 2005 UPC workshop at George Washington University and at SC 2005. This document is a result of those discussions.

# 2 Scope

1. This document describes UPC functions that supplement UPC.
2. All UPC language specifications as per V1.2 [2] are considered part of this specification, and therefore will not be addressed in this document.
3. Some parts of UPC Language Specifications V1.2 may be repeated for self-containment and clarity of the functions defined here.
4. This document extends the UPC Collective Operations Specification V1.0 [1, 4]. When the extensions are approved, these two documents will be merged to form a single document suitable for inclusion in the UPC Language Specification document.

# 3 Definitions

1. **Collective:**  
A requirement placed on some language operations which constrains invocations of such operations to be matched<sup>1</sup> across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.
2. **Single-valued:**  
An operand to a collective operation, which has the same value on every thread. The behavior of the operation is otherwise undefined.
3. **In-place:**  
A hint to a collective operation to perform it *in place*. This reduces the amount of duplicate data.
4. **Synchronous Collective Call:**  
Collective calls which block and wait until the corresponding collective operation is completed.
5. **Asynchronous Collective Call:**  
Collective calls which start a collective operation and return immediately, and must later be completed using a synchronization function. Only one outstanding asynchronous operation is allowed on a UPC collective handle at any given time.

---

<sup>1</sup>A collective operation need not provide any actual synchronization between threads, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely upon such unspecified synchronization for correctness.

## 4 Common Requirements

The following requirements apply to all of the functions defined in this document whose names begin with `upc_all...`.

1. All but the completion functions in this document are collective; therefore, must be called by all threads collectively.
2. Collective functions may not be called between `upc_notify` and the corresponding `upc_wait`.
3. The second to last argument of each collective function is the variable `mode` of type `upc_flag_t`. Values of `mode` are formed by performing a bitwise OR of a constant of the form `UPC_IN_XSYNC` and a constant of the form `UPC_OUT_YSYNC`, where  $X$  and  $Y$  may be NO, MY, or ALL.  
**Forward reference:** `upc_flag_t` (5.1.2).
4. If `mode` has the value  $(UPC\_IN\_XSYNC \mid UPC\_OUT\_YSYNC)$ , 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<sup>2</sup>

and if  $Y$  is

NO : the collective function may read or write data until the last thread has returned from the collective function call,

MY : the collective function may return in a thread only after all reads or writes of data with affinity to the thread are complete<sup>3</sup>, and

ALL : the collective function call may return only after all reads and writes of data are complete<sup>4</sup>

`UPC_IN_XSYNC` alone is equivalent to  $(UPC\_IN\_XSYNC \mid UPC\_OUT\_ALLSYNC)$

`UPC_OUT_XSYNC` alone is equivalent to  $(UPC\_IN\_ALLSYNC \mid UPC\_OUT\_XSYNC)$

and 0 is equivalent to  $(UPC\_IN\_ALLSYNC \mid UPC\_OUT\_ALLSYNC)$ , where  $X$  is NO, MY, or ALL.

5. The `UPC_PUSH` or the `UPC_PULL` flags may be OR-ed with the `mode` arguments to suggest to the implementation respectively that source thread(s) write or destination thread(s) read the data moved in the relocalization collectives. The implementation is free to interpret these flags in any way it chooses as long as the specification is not violated.
6. The `mode` flag allows specifying *in place* behavior of the collectives by OR-ing `UPC_IN_PLACE` with other defined bits.
7. The `mode` flag also allows specifying asynchronous behavior of the collectives by OR-ing `UPC_ASYNC` with other defined bits. In such a case, the semantics of the synchronization flags should be interpreted as follows: constraints that reference entry to a function call correspond to entering the posting call that initiate the asynchronous collective operation, and constraints that reference returning from a function call correspond to returning from the `upc_wait()` or successful

---

<sup>2</sup>`UPC_IN_ALLSYNC` requires the collective function to guarantee that after all threads have entered the collective function call, all threads will read the same values of the input data.

<sup>3</sup>`UPC_OUT_MYSYNC` requires the collective function to guarantee that after a thread returns from the collective function call, the thread will not read any earlier values of the output data with affinity to that thread.

<sup>4</sup>`UPC_OUT_ALLSYNC` requires the collective function to guarantee that after a thread returns from the collective function call, the thread will not read any earlier values of the output data. `UPC_OUT_ALLSYNC` is not required to provide an "implied" barrier. For example, if the entire collective operation has been completed by a certain thread before some other threads have reached their corresponding function calls, then that thread may exit its call.

`upc_test()` call that completes the asynchronous collective operation. Also note that all the mode flags that govern an asynchronous collective operation are passed to the library during the asynchronous posting call.

8. For the `upc_all_prefix_reduce` operation, if `UPC_EXCLUSIVE_PREFIX_REDUCE` is OR-ed with the mode arguments, then an exclusive prefix reduction is performed.
9. The last argument of each function is the variable `team` of type `upc_team_t`, which allows specifying a collective operation to work on a group of threads.

## 5 Extended Collectives Library

### 5.1 Standard header

1. The standard header is `upc_collective.h`
2. `upc_collective.h` defines `upc_flag_t` which is an integral type to the collective specification.
3. `upc_collective.h` defines `upc_team_t` which is an integral type to the collective specification.
4. `upc_collective.h` defines `upc_coll_handle_t` which is an integral type to the asynchronous collective specification.

### 5.2 Predefined Constants

1. `UPC_PUSH`, `UPC_PULL`:  
Or-ed with the *mode* operands to supply a hint to the implementation. User can choose either a *push* or *pull* implementation during a call to the function.
2. `UPC_IN_PLACE`:  
Substituted for the `src` argument to denote an *in place* operation<sup>5</sup>.
3. `UPC_ASYNC`:  
Or-ed with the *mode* operands to denote nonblocking collective function, which will have to be completed by a corresponding completion function.
4. `UPC_HANDLE_COMPLETE`:  
Used to indicate a completed or invalid handle. A successful completion function would set the corresponding handle to `UPC_HANDLE_COMPLETE`.
5. `UPC_EXCLUSIVE_PREFIX_REDUCE`:  
Or-ed with the *mode* operands in `upc_all_prefix_reduce` to perform an exclusive prefix reduction.  
**Forward Reference:** `upc_all_prefix_reduce` (5.5.1)
6. `UPC_TEAM_ALL`:  
If passed in as argument to `team`, all threads participate in the collective operation.

### 5.3 Extended Relocalization Operations

Based on the mode operands, the function returns either a handle (if `UPC_ASYNC` is present) or a `void`. If `UPC_ASYNC` is OR-ed with other arguments in the *mode*, then the function becomes a posting operation and returns a `upc_coll_handle_t`. This handle must then be consumed by a completion function (see 5.4).

---

<sup>5</sup>Duplicate copies of data can be easily eliminated for exchange, permute and prefix-reduce. However, for other collectives there is a certain degree of complexity in handling this feature due to how data layouts differ between `src` and `dst`.

### 5.3.1 The `upc_all_broadcast_x` function

#### Synopsis

```
1. #include <upc.h>
   #include <upc_collective_ext.h>

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

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

#### Description

1. The `upc_all_broadcast_x` function copies a block of data with affinity to a single thread to an arbitrary location on each destination thread.
2. The number of bytes in each block is `nbytes`, which must be strictly greater than 0; otherwise no copying will take place.
3. The function treats the `src` pointer as if it pointed to a shared memory area with type: `shared [] char [nelems]`
4. The function treats the `dst` pointer as if it pointed to a shared memory area with type: `shared [] char * shared [THREADS]`
5. The effect of the function is equivalent to copying `nbytes` elements of the array pointed to by `src` to a shared array pointed to by `dst[MYTHREAD]`. The behavior is undefined unless `upc_threadof( *dst[MYTHREAD] ) == MYTHREAD`

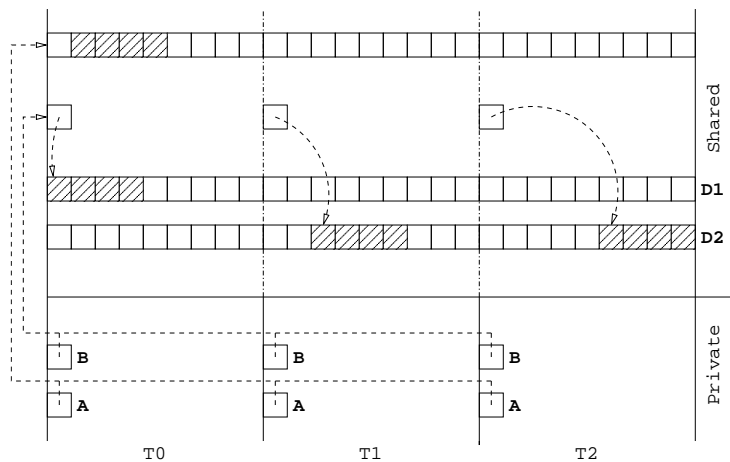


Figure 1: `upc_all_broadcast_x`

## Examples

1. The following example of `upc_all_broadcast_x` corresponds to figure 1:

```
#define BLK 9
#define NELEMS BLK*THREADS

shared [BLK] char A[NELEMS];
shared [] char * shared [1] B[THREADS];

shared [BLK] char D1[NELEMS];
shared [BLK] char D2[NELEMS];

... Initialize A ...

if( MYTHREAD == 0 )
{
    B[0] = (shared [] char *)&(D1[0]);
    B[1] = (shared [] char *)&(D2[11]);
    B[2] = (shared [] char *)&(D2[23]);
}

upc_all_broadcast_x( B, &A[1], 4*sizeof(char),
                    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL,
                    UPC_TEAM_ALL );
```

### 5.3.2 The `upc_all_scatter_x` function

#### Synopsis

```
1. #include <upc.h>
   #include <upc_collective_ext.h>

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

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

#### Description

1. The `upc_all_scatter_x` function allows varying blocks of data (possibly located in separate arrays) on the source thread to be copied to (possibly separate arrays on) the destination threads.
2. It allows for explicit addressing of the segments of the source array(s) on a single thread and the destination array(s) on all the threads. Both the source segments on a single thread and the destination segments on each thread can be in separate arrays.

- The `upc_all_scatter_x` function copies a specified block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the same or different thread. The number of bytes in each block is `nbytes[MYTHREAD]`.
- `nbytes` is a pointer to an array of `THREADS` integers, each representing the number of bytes in a block. Each element must be strictly greater than 0, otherwise copying will not take place for the corresponding thread.
- The `upc_all_scatter_x` function treats the `src` and `dst` pointers as if they pointed to a shared memory area with the type:  
`shared [] char * shared [THREADS]`

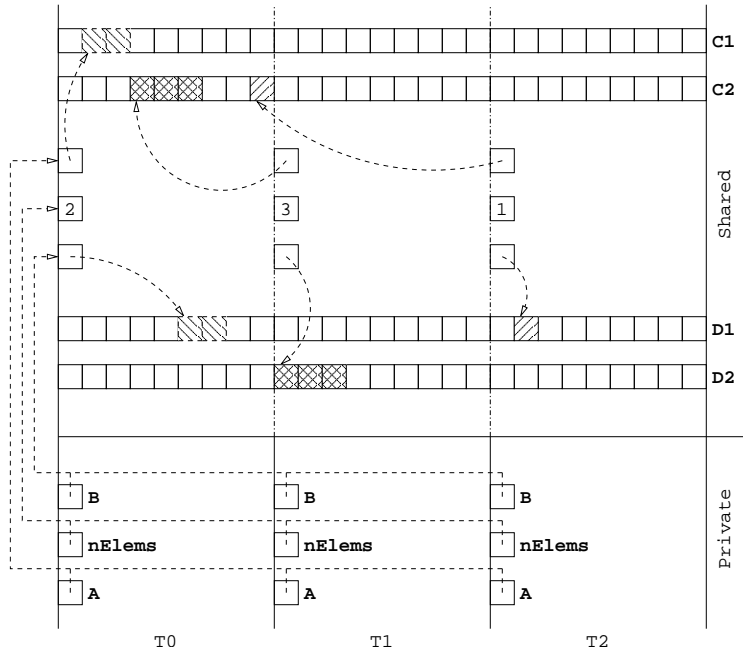


Figure 2: `upc_all_scatter_x`

## Examples

- The following example of `upc_all_scatter_x` corresponds to figure 2:

```
#define BLK 9
#define NELEMS BLK*THREADS

shared [] char * shared [1] A[THREADS];
shared [] char * shared [1] B[THREADS];

shared [1] size_t nBytes[THREADS];

shared [BLK] char C1[NELEMS];
shared [BLK] char C2[NELEMS];

shared [BLK] char D1[NELEMS];
shared [BLK] char D2[NELEMS];
```

```

... Initialize C1, C2 ...

if( MYTHREAD == 0 )
{
    nBytes[0] = 2*sizeof(char);
    nBytes[1] = 3*sizeof(char);
    nBytes[2] = 1*sizeof(char);

    A[0] = (shared [] char *)&(C1[1]);
    A[1] = (shared [] char *)&(C2[3]);
    A[2] = (shared [] char *)&(C2[8]);

    B[0] = (shared [] char *)&(D1[5]);
    B[1] = (shared [] char *)&(D2[9]);
    B[2] = (shared [] char *)&(D1[19]);
}

upc_all_scatter_x( B, A, nBytes,
                  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL,
                  UPC_TEAM_ALL );

```

### 5.3.3 The `upc_all_gather_x` function

#### Synopsis

```

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

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

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

```

#### Description

1. The `upc_all_gather_x` function allows a varying block of data (located possibly in separate arrays) to be copied from each thread to (possibly separate arrays on) a single thread.
2. It allows for explicit addressing of the source array(s) on all the threads and the segments of destination array(s) on a single thread. Both the source segments on each thread and the destination segments on a single thread can be in separate arrays.
3. The `upc_all_gather_x` function copies a block of shared memory that has affinity to the  $i$ th thread to a specified block of a shared memory area that has affinity to a single thread. The number of bytes in each block is `nbytes[i]`.

4. `nbytes` is a pointer to an array of `THREADS` integers, each representing the number of bytes in a block. Each element must be strictly greater than 0, otherwise copying of a block from the corresponding thread will not take place.
5. The `upc_all_gather_x` function treats the `src` and `dst` pointers as if they pointed to a shared memory area with the type:
 

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

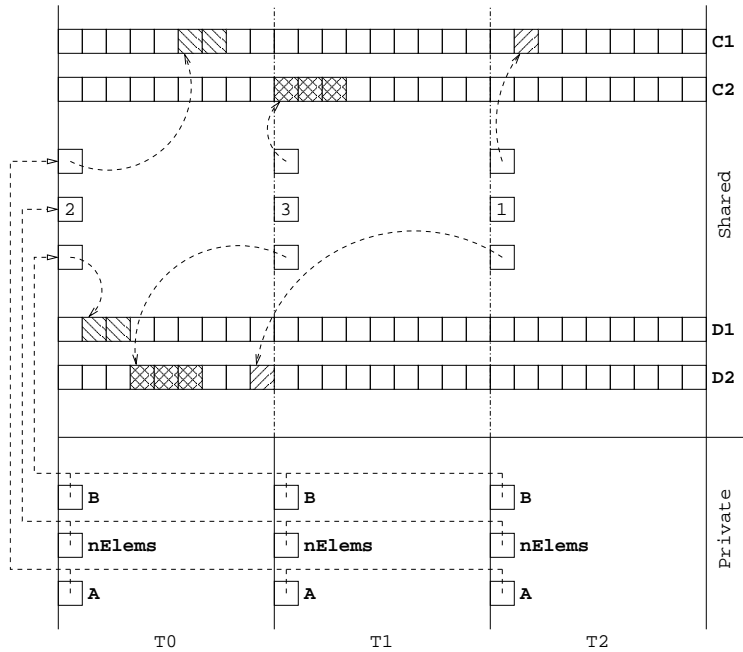


Figure 3: `upc_all_gather_x`

## Examples

1. The following example of `upc_all_gather_x` corresponds to figure 3::

```
#define BLK 9
#define NELEMS BLK*THREADS

shared [] char * shared [1] A[THREADS];
shared [] char * shared [1] B[THREADS];

shared [1] size_t nBytes[THREADS];

shared [BLK] char C1[NELEMS];
shared [BLK] char C2[NELEMS];

shared [BLK] char D1[NELEMS];
shared [BLK] char D2[NELEMS];

... Initialize C1, C2 ...

if( MYTHREAD == 0 )
```

```

{
    nBytes[0] = 2*sizeof(char);
    nBytes[1] = 3*sizeof(char);
    nBytes[2] = 1*sizeof(char);

    A[0] = (shared [] char *)&(C1[5]);
    A[1] = (shared [] char *)&(C2[9]);
    A[2] = (shared [] char *)&(C1[19]);

    B[0] = (shared [] char *)&(D1[1]);
    B[1] = (shared [] char *)&(D2[3]);
    B[2] = (shared [] char *)&(D2[8]);
}

upc_all_gather_x( B, A, nBytes,
                 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH,
                 UPC_TEAM_ALL );

```

### 5.3.4 The `upc_all_gather_all_x` function

#### Synopsis

```

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

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

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

```

#### Description

1. The `upc_all_gather_all_x` function allows a varying block of data (located possibly in separate arrays) to be copied from each thread to (possibly separate arrays on) all the threads. It can be thought of as `upc_all_gather_x` except that all threads copy the data instead of just one.
2. It allows for explicit addressing of the segments of the source array(s) and the segments of the destination array(s) on all threads. Both the source segments and the destination segments on each thread can be in separate arrays.
3. The `upc_all_gather_all_x` function copies a block of shared memory from one shared memory area with affinity to the  $i$ th thread to the  $i$ th block of a shared memory area on each thread. The number of elements in each block is `nbytes[i]`.
4. `nbytes` is a pointer to an array of `THREADS` integers, each representing the number of bytes in a block. Each array element must be strictly greater than 0, otherwise copying of a block from the

corresponding thread will not take place.

- The `upc_all_gather_all_x` function treats the `src` pointer as if it pointed to a shared memory area with the type:
 

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

 and the `dst` pointer as if it pointed to a shared memory area with the type:
 

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

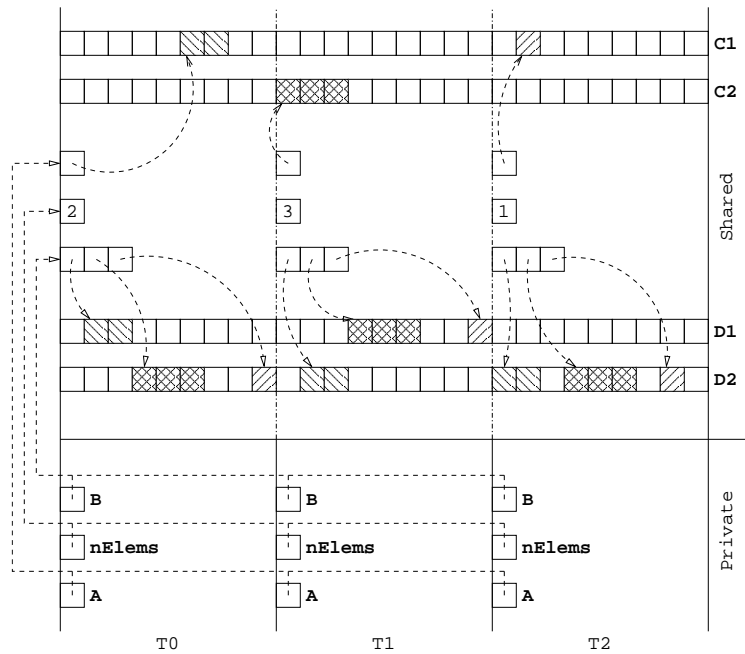


Figure 4: `upc_all_gather_all_x`

## Examples

- The following example of `upc_all_gather_all_x` corresponds to figure 4:

```
#define BLK 9
#define NELEMS BLK*THREADS

shared [] char * shared [1] A[THREADS];
shared [] char * shared [THREADS] B[THREADS*THREADS];

shared [1] size_t nBytes[THREADS];

shared [BLK] char C1[NELEMS];
shared [BLK] char C2[NELEMS];

shared [BLK] char D1[NELEMS];
shared [BLK] char D2[NELEMS];

... Initialize C1, C2 ...

if( MYTHREAD == 0 )
```

```

{
    nBytes[0] = 2*sizeof(char);
    nBytes[1] = 3*sizeof(char);
    nBytes[2] = 1*sizeof(char);

    A[0] = (shared [] char *)&(C1[5]);
    A[1] = (shared [] char *)&(C2[9]);
    A[2] = (shared [] char *)&(C1[19]);

    B[0] = (shared [] char *)&(D1[1]);
    B[1] = (shared [] char *)&(D2[3]);
    B[2] = (shared [] char *)&(D2[8]);
    B[3] = (shared [] char *)&(D2[10]);
    B[4] = (shared [] char *)&(D1[12]);
    B[5] = (shared [] char *)&(D1[17]);
    B[6] = (shared [] char *)&(D2[18]);
    B[7] = (shared [] char *)&(D2[21]);
    B[8] = (shared [] char *)&(D2[25]);
}

upc_all_gather_all_x( B, A, nBytes,
                    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PUSH,
                    UPC_TEAM_ALL );

```

### 5.3.5 The `upc_all_exchange_x` function

#### Synopsis

```

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

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

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

```

#### Description

1. The `upc_all_exchange_x` function allows a varying block of data (located possibly in separate arrays) to be copied from each thread to (possibly separate arrays on) threads corresponding to the block number.
2. It allows for explicit addressing of the segments of source array(s) and the segments of destination array(s) on all threads. Both the source segments and the destination segments on each thread can be in separate arrays.

3. `nbytes` is a pointer to an array of `THREADS` integers, each representing the number of bytes in a block. Each array element must be strictly greater than 0, otherwise copying of a block in the corresponding thread will not take place.
4. The `upc_all_exchange_x` function treats the `src` and `dst` pointers as if they pointed to a shared memory area with the type:
 

```
shared [] char * shared [THREADS] [THREADS*THREADS]
```
5. For each pair of threads  $i$  and  $j$ , the effect is equivalent to copying the  $i$ th block of `nbytes[i]` bytes that has affinity to thread  $j$  pointed to by `src[i]` to the  $j$ th block of `nbytes[i]` bytes that has affinity to thread  $i$  pointed to by `dst[i]`.

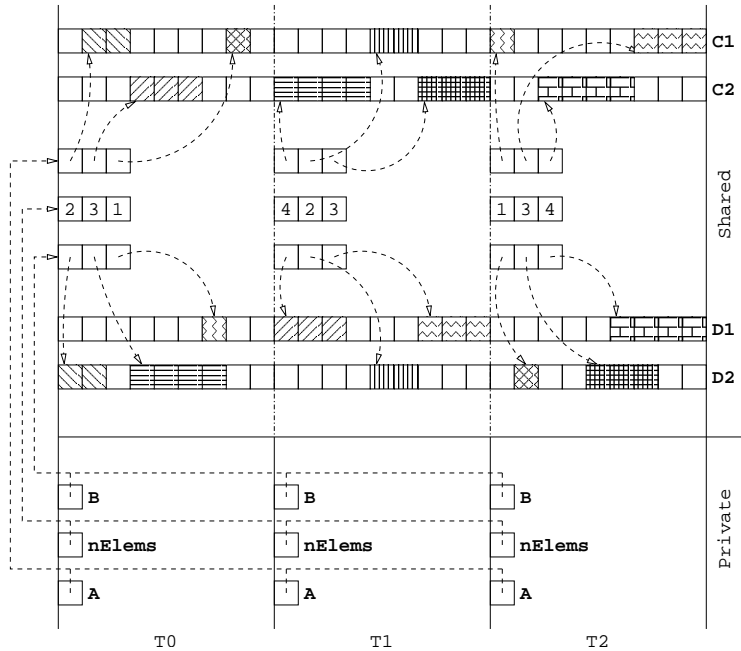


Figure 5: `upc_all_exchange_x`

## Examples

1. The following example of `upc_all_exchange_x` corresponds to figure 5:

```
#define BLK 9
#define NELEMS BLK*THREADS

shared [] char * shared [THREADS] A[THREADS*THREADS];
shared [] char * shared [THREADS] B[THREADS*THREADS];

shared [THREADS] size_t nbytes[THREADS*THREADS];

shared [BLK] char C1[NELEMS];
shared [BLK] char C2[NELEMS];

shared [BLK] char D1[NELEMS];
shared [BLK] char D2[NELEMS];
```

```

... Initialize C1, C2 ...

if( MYTHREAD == 0 )
{
    nBytes[0] = 2*sizeof(char);
    nBytes[1] = 3*sizeof(char);
    nBytes[2] = 1*sizeof(char);
    nBytes[3] = 4*sizeof(char);
    nBytes[4] = 2*sizeof(char);
    nBytes[5] = 3*sizeof(char);
    nBytes[6] = 1*sizeof(char);
    nBytes[7] = 3*sizeof(char);
    nBytes[8] = 4*sizeof(char);

    A[0] = (shared [] char *)&(C1[1]);
    A[1] = (shared [] char *)&(C2[3]);
    A[2] = (shared [] char *)&(C1[7]);
    A[3] = (shared [] char *)&(C2[9]);
    A[4] = (shared [] char *)&(C1[13]);
    A[5] = (shared [] char *)&(C2[15]);
    A[6] = (shared [] char *)&(C1[18]);
    A[7] = (shared [] char *)&(C1[24]);
    A[8] = (shared [] char *)&(C2[20]);

    B[0] = (shared [] char *)&(D2[0]);
    B[1] = (shared [] char *)&(D2[3]);
    B[2] = (shared [] char *)&(D1[6]);
    B[3] = (shared [] char *)&(D1[9]);
    B[4] = (shared [] char *)&(D2[13]);
    B[5] = (shared [] char *)&(D1[15]);
    B[6] = (shared [] char *)&(D2[19]);
    B[7] = (shared [] char *)&(D2[22]);
    B[8] = (shared [] char *)&(D1[23]);
}

upc_all_exchange_x( B, A, nBytes,
                    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL,
                    UPC_TEAM_ALL );

```

### 5.3.6 The `upc_all_permute_x` function

#### Synopsis

```

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

void upc_all_permute_x( shared void * shared * dst,
                       shared const void * shared * src,
                       shared size_t * perm,
                       shared size_t * nbytes,
                       upc_flag_t mode,

```

```

        upc_team_t team );

upc_coll_handle_t upc_all_permute_x( shared void * shared * dst,
                                     shared const void * shared * src,
                                     shared size_t * perm,
                                     shared size_t * nbytes,
                                     upc_flag_t mode,
                                     upc_team_t team );

```

## Description

1. The `upc_all_permute_x` function allows a varying block of data (located possibly in separate arrays) to be copied from one thread to another (possibly in separate arrays).
2. It allows for explicit addressing of the segments of source array(s) and the segments of destination array(s) on all threads. Both the source segments and the destination segments on each thread can be in separate arrays.
3. `nbytes` is a pointer to an array of `THREADS` integers, each representing the number of bytes in a block. Each array element must be strictly greater than 0, otherwise copying of a block in the corresponding thread will not take place.
4. `perm` is a pointer to an array of `THREADS` integers, each denoting the thread to which the data is to be copied to.
5. The `upc_all_permute_x` function treats the `src` and `dst` pointers as if they pointed to a shared memory area with the type:

```
shared [] char * shared [THREADS]
```
6. For each thread  $i$ , the effect is equivalent to copying the block of `nbytes[i]` bytes pointed to by `src[i]` that has affinity to thread  $i$  to a specified block of `nbytes[i]` bytes pointed to by `dst[perm[i]]` that has affinity to thread `perm[i]`.

## Examples

1. The following example of `upc_all_permute_x` corresponds to figure 6:

```

#define BLK 9
#define NELEMS BLK*THREADS

shared [] char * shared [1] A[THREADS];
shared [] char * shared [1] B[THREADS];

shared [1] int P[THREADS];
shared [1] size_t nBytes[THREADS];

shared [BLK] char C1[NELEMS];
shared [BLK] char C2[NELEMS];

shared [BLK] char D1[NELEMS];
shared [BLK] char D2[NELEMS];

... Initialize C1, C2 ...

```

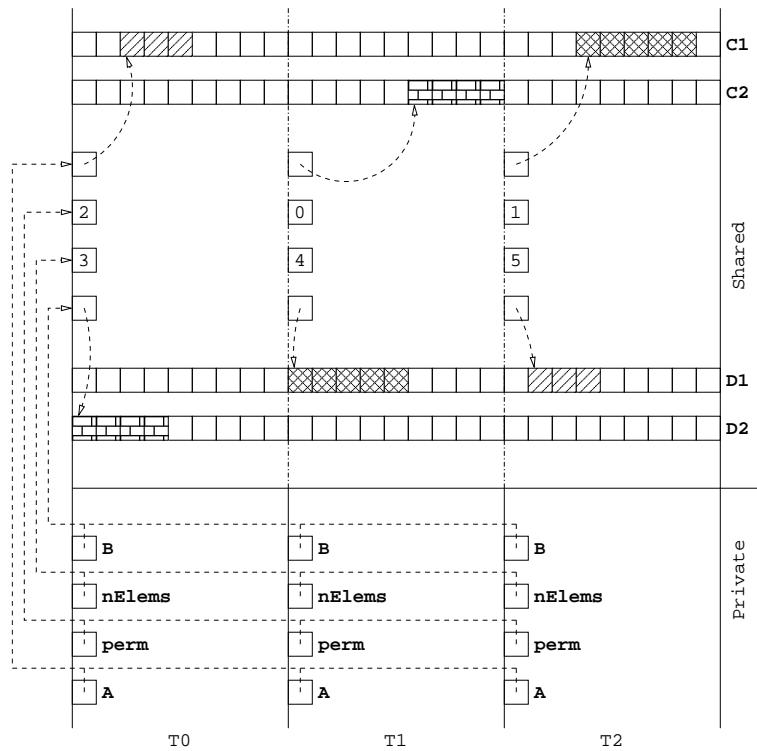


Figure 6: upc\_all\_permute\_x

```

if( MYTHREAD == 0 )
{
    P[0] = 2;
    P[1] = 0;
    P[2] = 1;

    nBytes[0] = 3*sizeof(char);
    nBytes[1] = 4*sizeof(char);
    nBytes[2] = 5*sizeof(char);

    A[0] = (shared [] char *)&(C1[2]);
    A[1] = (shared [] char *)&(C2[14]);
    A[2] = (shared [] char *)&(C1[21]);

    B[0] = (shared [] char *)&(D2[0]);
    B[1] = (shared [] char *)&(D1[9]);
    B[2] = (shared [] char *)&(D1[19]);
}

upc_all_permute_x( B, A, P, nBytes,
                  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL,
                  UPC_TEAM_ALL );

```

## 5.4 Completion Operations

### 5.4.1 The `upc_wait` function

#### Synopsis

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

   void upc_wait( upc_coll_handle_t handle );
```

#### Description

1. `upc_wait` returns when the operation identified by `handle` is completed. It then sets the handle to `UPC_HANDLE_COMPLETE`.
2. If `UPC_HANDLE_COMPLETE` is passed in as argument to `handle`, then the function returns immediately (basically, a no-op).

### 5.4.2 The `upc_test` function

#### Synopsis

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

   int upc_test( upc_coll_handle_t handle );
```

#### Description

1. `upc_test` returns a positive integer if the operation identified by `handle` is complete. It then sets the handle to `UPC_HANDLE_COMPLETE`. Otherwise, it returns a negative integer.
2. `upc_test` returns successfully exactly in those situations where `upc_wait` returns.
3. The use of `upc_test` allows scheduling alternative activities within a single thread of execution.

### 5.4.3 The `upc_waitany` function

#### Synopsis

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

   int upc_waitany( int count, upc_coll_handle_t * handles );
```

#### Description

1. The `upc_waitany` function is used to complete one out of several asynchronous collective operations.
2. It blocks until one of the operations associated with the handles has completed.
3. The function returns the array index of the completed handle and sets the appropriate handle from the array to `UPC_HANDLE_COMPLETE`.

4. If more than one operation can be completed, then `upc_waitany` arbitrarily picks one operation and completes it.

#### 5.4.4 The `upc_testany` function

##### Synopsis

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

   int upc_testany( int count, upc_coll_handle_t * handles );
```

##### Description

1. The `upc_testany` function is used to test one out of several asynchronous collective operations for completion.
2. It returns immediately whether any of the handles have completed or not.
3. If one of the handles is complete, it returns the index of the handle, sets the handle at that index to `UPC_HANDLE_COMPLETE`. Otherwise, a negative number is returned.
4. If more than one operation can be completed, then `upc_testany` arbitrarily picks one operation and completes it.

#### 5.4.5 The `upc_waitall` function

##### Synopsis

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

   void upc_waitall( int count, upc_coll_handle_t * handles );
```

##### Description

1. The `upc_waitall` function is used to complete all operations in an array.
2. It blocks until all handles in the array are complete.
3. Once a handle is completed, the associated index in the `handles` array is set to `UPC_HANDLE_COMPLETE`.

#### 5.4.6 The `upc_testall` function

##### Synopsis

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

   int upc_testall( int count, upc_coll_handle_t * handles );
```

## Description

1. The `upc_testall` function is used to test all operations in an array for completion.
2. If all operations have completed, it returns a positive integer, sets the corresponding handles to `UPC_HANDLE_COMPLETE`. Otherwise, a negative integer is returned. In this case, no handle is modified.

### 5.4.7 The `upc_waitsome` function

#### Synopsis

1. 

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

int upc_waitsome( int count, upc_coll_handle_t * handles );
```

## Description

1. The `upc_waitsome` function is used to complete all enabled operations in an array.
2. It waits until at least one of the operations is complete. Once this happens it goes on completing that operation as well as all others that are complete. The handles in the appropriate indexes are set to `UPC_HANDLE_COMPLETE`.
3. It returns a count of completed handles.

### 5.4.8 The `upc_testsome` function

#### Synopsis

1. 

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

int upc_testsome( int count, upc_coll_handle_t * handles );
```

## Description

1. The `upc_testsome` function is used to test for completion of all enabled operations in a list. It behaves like `upc_waitsome`, except that it returns immediately.
2. If some operations have completed it returns the number. If none has completed then it returns 0.

### 5.4.9 The `upc_get_status` function

#### Synopsis

1. 

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

int upc_get_status( upc_coll_handle_t handle );
```

## Description

1. This is useful in cases when status information is needed without actually completing any of the asynchronous collective operations.
2. If the operation indicated by the `handle` is complete, the function returns a positive integer; otherwise, it simply returns a negative integer.
3. A subsequent call to `upc_wait` or `upc_test` is required to complete the operation.

## 5.5 Extended Computational Operations

An integral part of the computational collective operations is `upc_op_t`. A variable of type `upc_op_t` can have the following values:

<code>UPC_ADD</code>	Addition.
<code>UPC_MULT</code>	Multiplication.
<code>UPC_AND</code>	Bitwise AND for integer and character variables. Results are undefined for floating point and complex types.
<code>UPC_OR</code>	Bitwise OR for integer and character variables. Results are undefined for floating point and complex types.
<code>UPC_XOR</code>	Bitwise XOR for integer and character variables. Results are undefined for floating point and complex types.
<code>UPC_LOGAND</code>	Logical AND for all variable types.
<code>UPC_LOGOR</code>	Logical OR for all variable types.
<code>UPC_MIN</code>	For all data types, find the minimum value.
<code>UPC_MAX</code>	For all data types, find the maximum value.
<code>UPC_FUNC</code>	Use the specified commutative function <code>func</code> to operate on the data in the <code>src</code> array at each step.
<code>UPC_NONCOMM_FUNC</code>	Use the specified non-commutative function <code>func</code> to operate on the data in the <code>src</code> array at each step.

### 5.5.1 The `upc_all_reduce_all` function and Exclusive Prefix-reduction

#### Synopsis

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

void upc_all_reduce_allT( shared void * restrict dst,
                        shared const void * restrict src,
                        upc_opt_t op,
                        size_t nelems,
                        size_t blk_size,
                        TYPE(*func)( TYPE, TYPE ),
                        upc_flag_t mode );

void upc_all_prefix_reduceT( shared void * restrict dst,
                           shared const void * restrict src,
                           upc_opt_t op,
                           size_t nelems,
                           size_t blk_size,
                           TYPE(*func)( TYPE, TYPE ),
                           upc_flag_t mode );
```

## Description

1. The function prototype above represents 17 variants of the `upc_all_reduce_allT` function, where  $T$  and  $TYPE$  have the following correspondences:<sup>6</sup>

$T$	$TYPE$	$T$	$TYPE$
C	signed char	D	double
UC	unsigned char	LD	long double
S	signed short	LL	long long
US	unsigned short	ULL	unsigned long long
I	signed int	CX	_Complex float
UI	unsigned int	DX	_Complex double
L	signed long	LDX	_Complex long double
UL	unsigned long	B	_Bool
F	float		

2. On completion of the `upc_all_reduce_all` variants, the value of the  $TYPE$  shared object referenced by `dst[i]` is `src[0]⊕src[1]⊕...⊕src[nelems-1]` for all  $i$ , where  $0 \leq i < \text{THREADS}$  and  $\oplus$  is the operator specified by the variable `op`.<sup>7</sup>
3. On completion of the `upc_all_prefix_reduce` variants, the value of  $TYPE$  shared object referenced by `dst[i]` depends on the value of `mode`. If the value of `(mode & UPC_EXCLUSIVE_PREFIX_REDUCE)` is zero, then the value is `src[0]⊕src[1]⊕...⊕src[i]` for  $0 \leq i \leq \text{nelems}-1$ , where  $\oplus$  is the operator specified by the variable `op`. Otherwise, the value referenced by `dst[0]` is undefined and `dst[i]` for  $0 < i \leq \text{nelems}-1$  is `src[0]⊕src[1]⊕...⊕src[i-1]`.
4. If the value of `blk_size` passed to these functions is greater than 0, then they treat `src` pointer as if it pointed to a shared memory area of `nelems` elements of type  $TYPE$  and blocking factor `blk_size`, and therefore had type:  
`share [blk_size] TYPE [nelems]`
5. If the value `blk_size` passed to these functions is 0, then they treat the `src` pointer as if it pointed to a shared memory area of `nelems` elements of the  $TYPE$  with an indefinite layout qualifier, and therefore had type<sup>8</sup>:  
`share [] TYPE [nelems]`
6. The phase of the `src` pointer is respected when referencing array elements, as specified above.
7. The `upc_all_prefix_reduceT` treats the `dst` pointer equivalently to the `src` pointer as described in the past 3 paragraphs.
8. `upc_all_prefix_reduceT` requires the affinity and phase of the `src` and the `dst` pointers to match<sup>9</sup>.
9. The `upc_all_reduce_allT` treats the `dst` pointer as having type:  
`shared [1] TYPE[THREADS]`
10. `upc_all_reduce_allT` requires the `dst` pointer to have affinity to thread 0.
11. `upc_all_reduce_allT` treats the `dst` pointer as if it has phase 0.
12. For type `_Bool`, the only valid operations are `UPC_LOGAND`, `UPC_LOGOR`, `UPC_FUNC`, and `UPC_NONCOMM_FUNC`.

<sup>6</sup>For example, if  $T$  is C, then  $TYPE$  must be signed char.

<sup>7</sup>All elements `dst[0]...dst[THREADS-1]` shall compare equal, even in the presence of floating point roundoff or similar sources of lack of associativity of built-in functions.

<sup>8</sup>Note that `upc_blocksize(src) == 0` if `src` has this type, so the argument value of 0 has a natural connection to the block size of `src`.

<sup>9</sup>*i.e.*, `upc_threadof(src) == upc_threadof(dst) && upc_phaseof(src) == upc_phaseof(dst)`.

## References

- [1] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, 2005.
- [2] UPC Language Specifications V1.2. [online], June 2005.  
[http://www.gwu.edu/~upc/docs/upc\\_specs\\_1.2.pdf](http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf).
- [3] E. Wiebel and D. Greenberg. Collective operations in UPC. Technical Report CCS TR-02-159, IDA Center for Computing Sciences, March 2002.
- [4] 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](http://www.gwu.edu/~upc/docs/UPC_Coll_Spec_V1.0.pdf).