Specifications for UPC’s
Extended Collective Operations

Draft version 0.1 for 2005 UPC Workshop

Department of Computer Science
Michigan Technological University
Houghton, MI 49931

September 17, 2005
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>upc_all_broadcast_x</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>upc_all_scatter_x</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>upc_all_gather_x</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>upc_all_gather_all_x</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>upc_all_exchange_x</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>upc_all_permute_x</td>
<td>15</td>
</tr>
</tbody>
</table>
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 and May 2003.

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. This document is a result of that discussion.

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\(^1\) 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:
   The syntax and semantic for the in place flag are in progress.

\(^1\) 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 of the functions in this document are collective.

2. All collective function arguments are single-valued.

3. Collective functions may not be called between upc_notify and the corresponding upc_wait.

4. The last argument of each collective function is the variable mode of type upc_flag_t. Values of mode are formed by or-ing together a constant of the form UPC_IN_XSYNC, a constant of the form UPC_OUT_YSYNC, and a constant of the form UPC_IN_PLACE, where X and Y may be NO, MY, or ALL.

   Forward reference: upc_flag_t (6.1.2).

5. If mode has the value (UPC_IN_XSYNC | UPC_OUT_YSYNC), then if X is

   NO: the collective function may begin to read or write data when the first thread has entered the collective function call,
   MY: the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call, and
   ALL: the collective function may begin to read or write data only after all threads have entered the collective function call

   and if Y is

   NO: the collective function may read 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, and
   ALL: the collective function call may return only after all reads and writes of data are complete

   UPC_IN_XSYNC alone is equivalent to (UPC_IN_XSYNC | UPC_OUT_ALLSYNC)
   UPC_OUT_XSYNC alone is equivalent to (UPC_IN_ALLSYNC | UPC_OUT_XSYNC)
   and 0 is equivalent to (UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC), where X is NO, MY, or ALL.

---

2UPC_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.

3UPC_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.

4UPC_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.
6. In the examples of this document, the in place option is specified by or-ing UPC_IN_PLACE with the values of mode.

7. The UPC_PUSH or the UPC_PULL flags may be or-ed with the mode to suggest to the implementation that source thread(s) write or destination thread(s) read, respectively, 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.

8. The mode flag allows specifying asynchronous behavior of the collectives by or-ing UPC_ASYNC with other defined bits.

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 specifications.

5.2 Extended Relocalization Operations

5.2.1 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 nelems,
                            size_t typesize,
                            upc_flag_t mode );

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 elements in each block is nelems, 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 \textit{typesize} is the size of the data type in bytes.

6. The effect of the function is equivalent to copying \textit{nelems} elements of the array pointed to by \textit{src} to a shared array pointed to by \textit{dst[MYTHREAD]}. The behavior is undefined unless

\begin{verbatim}
upc_threadof( *dst[MYTHREAD] ) == MYTHREAD
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{upc_all_broadcast_x}
\caption{upc\textunderscore all\textunderscore broadcast\_x}
\end{figure}

\textbf{Examples}

1. The following example of \texttt{upc\textunderscore all\textunderscore broadcast\_x} corresponds to figure 1:

\begin{verbatim}
#define BLK 9
#define NELEMS BLK\textstar 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\textunderscore barrier;
\end{verbatim}
5.2.2 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 * nelems,
                           size_t * typesize,
                           upc_flag_t mode );

**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.

3. 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 `nelems[MYTHREAD]`.

4. `nelems` is a pointer to an array of THREADS integers, each representing the number of elements in a block. Each element must be strictly greater than 0, otherwise copying will not take place for the corresponding thread.

5. 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]
   ```

6. The `typesize` is the size of the data type in bytes.

**Examples**

1. 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 nElems[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 ) {
    nElems[0] = 2;
nElems[1] = 3;
nElems[2] = 1;

    A[0] = (shared [] char *)&(C1[1]);
    A[1] = (shared [] char *)&(C2[3]);
B[0] = (shared [] char *)&(D1[5]);
B[1] = (shared [] char *)&(D2[9]);
B[2] = (shared [] char *)&(D1[19]);
}

upc_barrier;
upc_all_scatter_x((shared void * shared *)B,
  (shared const void * shared *)A, nElems, sizeof(char),
  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_IN_PLACE | UPC_PULL );
upc_barrier;

5.2.3 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 * nelems,
                           size_t typesize,
                           upc_flag_t mode );

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 ith thread to a specified block of a shared memory area that has affinity to a single thread. The number of elements in each block is nelems[i].

4. nelems is a pointer to an array of THREADS integers, each representing the number of elements 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]

6. The typesize is the size of the data type in bytes.
Examples

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

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

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

shared [1] size_t nElems[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 )
{
    nElems[0] = 2;
    nElems[1] = 3;
    nElems[2] = 1;
}
```
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_barrier;
upc_all_gather_x( (shared void * shared *)B,
    (shared const void * shared *)A, nElems, sizeof(char),
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_IN_PLACE | UPC_PUSH );
upc_barrier;

5.2.4 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,
                              size_t * nelems,
                              size_t typesize,
                              upc_flag_t mode );

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 $\text{nelems}[i]$.

4. $\text{nelems}$ is a pointer to an array of THREADS integers, each representing the number of elements 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.
5. The `upc_all_gather_all_x` function treats the `src` pointer as if it pointed to a shared memory area with the type:

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

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

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

6. The `typedef` is the size of the data type in bytes.

![Diagram of `upc_all_gather_all_x`](image)

**Figure 4: upc_all_gather_all_x**

**Examples**

1. The following example of `upc_all_gather_all_x` corresponds to figure 4:

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

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

shared [1] size_t nElems[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 )
{
    nElems[0] = 2;
    nElems[1] = 3;
    nElems[2] = 1;

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

upc_barrier;
upc_all_gather_all_x( (shared void * shared *)B,
    (shared const void * shared *)A, nElems, sizeof(char),
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_IN_PLACE | UPC_PUSH );
upc_barrier;

5.2.5 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 * nelems,
                       size_t typesize,
                       upc_flag_t mode );
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. nelems is a pointer to an array of THREADS integers, each representing the number of elements 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. The typesize is the size of the data type in bytes.

6. For each pair of threads $i$ and $j$, the effect is equivalent to copying the $i$th block of $\text{nelems}[i]$ elements that has affinity to thread $j$ pointed to by $\text{src}[i]$ to the $j$th block of $\text{nelems}[i]$ elements that has affinity to thread $i$ pointed to by $\text{dst}[i]$.

![Diagram](image_url)

**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 nElems[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 )
{
    nElems[0] = 2;
    nElems[1] = 3;
    nElems[2] = 1;
    nElems[3] = 4;
    nElems[4] = 2;
    nElems[5] = 3;
    nElems[6] = 1;
    nElems[7] = 3;
    nElems[8] = 4;

    A[0] = (shared [] char *)&(C1[1]);
    A[1] = (shared [] char *)&(C2[3]);
    A[2] = (shared [] char *)&(C1[7]);
    A[6] = (shared [] char *)&(C1[18]);
    A[8] = (shared [] char *)&(C2[20]);

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

upc_barrier;
upc_all_exchange_x( (shared void * shared *)B,
    (shared const void * shared *)A,
    (shared void *)nElems, sizeof(char),
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_IN_PLACE | UPC_PULL );
upc_barrier;

5.2.6 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 * nelems,
                           size_t typesize,
                           upc_flag_t mode );

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. nelems is a pointer to an array of THREADS integers, each representing the number of
   elements 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. The typesize is the size of the data type in bytes.
7. For each thread $i$, the effect is equivalent to copying the block of $\text{nelems}[i]$ elements pointed to by $\text{src}[i]$ that has affinity to thread $i$ to a specified block of $\text{nelems}[i]$ elements pointed to by $\text{dst}[\text{perm}[i]]$ that has affinity to thread $\text{perm}[i]$.

Figure 6: upc_all_permute_x

Examples

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

```c
#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 nElems[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 )
{
    P[0] = 2;
    P[1] = 0;
    P[2] = 1;
    nElems[0] = 3;
    nElems[1] = 4;
    nElems[2] = 5;
    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_barrier;
upc_all_permute_x((shared void * shared *)B,
  (shared const void * shared *)A,
  P, nElems, sizeof(char),
  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC | UPC_PULL );
upc_barrier;

6 Asynchronous Collectives Library

6.1 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 specifications.

3. <upc_collective.h> defines upc_handle_t which is an integral type to the asynchronous collective specifications, namely for the posting functions.

4. <upc_collective.h> defines upc_status_t which is an integral type to the asynchronous collective specifications, namely for the completion functions.
6.2 Reference constants

1. **UPC_ASYNC**:
   Or-ed with the *mode* operands to denote nonblocking collective function, which will have to be completed by a corresponding completion function.

2. **UPC_STATUS_IGNORE**:
   If passed in for the *status* argument to the completion functions, the status information is ignored.

3. **UPC_HANDLE_COMPLETE**:
   Used to indicate a completed or invalid handle. A successful completion function would set the corresponding handle to **UPC_HANDLE_COMPLETE**.

6.3 Posting Operations

The posting function prototypes are the same as the standard and extended functions with just one exception, they return a **upc_handle_t**, not a **void**. All description remain the same.

6.3.1 The **upc_all_broadcast** function

**Synopsis**

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

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

6.3.2 The **upc_all_broadcast_x** function

**Synopsis**

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

   upc_handle_t upc_all_broadcast_x( shared void * shared * dst,  
       shared const void * src,  
       size_t nelems,  
       size_t typesize,  
       upc_flag_t mode );
```  

6.3.3 The **upc_all_scatter** function

**Synopsis**

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

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

6.3.4 The **upc_all_scatter_x** function

Synopsis

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

   upc_handle_t upc_all_scatter_x( shared void * shared * dst,  
                                   shared const void * shared * src,  
                                   shared size_t * nelems,  
                                   size_t * typesize,  
                                   upc_flag_t mode );

6.3.5 The **upc_all_gather** function

Synopsis

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

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

6.3.6 The **upc_all_gather_x** function

Synopsis

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

   upc_handle_t upc_all_gather_x( shared void * shared * dst,  
                                  shared const void * shared * src,  
                                  shared size_t * nelems,  
                                  size_t typesize,  
                                  upc_flag_t mode );
6.3.7 The `upc_all_gather_all` function

Synopsis

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

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

6.3.8 The `upc_all_gather_all_x` function

Synopsis

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

   upc_status_t upc_all_gather_all_x( shared void * shared * dst,
                                      shared const void * shared * src,
                                      shared size_t * nelems,
                                      size_t typesize,
                                      upc_flag_t mode );
```

6.3.9 The `upc_all_exchange` function

Synopsis

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

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

6.3.10 The `upc_all_exchange_x` function

Synopsis

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

   upc_handle_t upc_all_exchange_x( shared void * shared * dst,
                                     shared const void * shared * src,
                                     shared size_t * nelems,
```
size_t typesize,
upc_flag_t mode );

6.3.11 The upc_all_permute function

Synopsis

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

data_t typesize,
upc_flag_t mode );

6.3.12 The upc_all_permute_x function

Synopsis

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

6.4 Completion Operations

6.4.1 The upc_wait function

Synopsis

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

Description

1. upc_wait returns when the operation identified by handle is completed. It then sets the handle to UPC_HANDLE_COMPLETE.
2. The function provides information of the completed operation in \texttt{status}, unless \texttt{UPC\_STATUS\_IGNORE} is passed in as argument.

3. If \texttt{UPC\_HANDLE\_COMPLETE} is passed in as argument to \texttt{handle}, then the function returns immediately.

6.4.2 The \texttt{upc\_test} function

\textbf{Synopsis}

1. \begin{verbatim}
   #include <upc.h>
   #include <upc_collective.h>

   int upc_test( upc_handle_t * handle,
                  upc_status_t * status );
\end{verbatim}

\textbf{Description}

1. \texttt{upc\_test} returns a positive integer if the operation identified by \texttt{handle} is complete. It then sets the handle to \texttt{UPC\_HANDLE\_COMPLETE} and passes information of the completed operation in \texttt{status}, unless \texttt{UPC\_STATUS\_IGNORE} is passed in as argument. Otherwise, it returns a negative integer.

2. \texttt{upc\_test} returns successfully exactly in those situations where \texttt{upc\_wait} returns. In that case, both return the same information.

3. The use of \texttt{upc\_test} allows scheduling alternative activities within a single thread of execution.

6.4.3 The \texttt{upc\_waitany} function

\textbf{Synopsis}

1. \begin{verbatim}
   #include <upc.h>
   #include <upc_collective.h>

   int upc_waitany( int count,
                   upc_handle_t * handles,
                   upc_status_t * status );
\end{verbatim}

\textbf{Description}

1. The \texttt{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 index of the handle that is completed in the array. The appropriate handle from the array is set to UPC_HANDLE_COMPLETE and status is recorded in status unless UPC_STATUS_IGNORE is passed in.

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

6.4.4 The upc_testany function

Synopsis

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

   int upc_testany( int count,
                   upc_handle_t * handles,
                   upc_status_t * status );

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 and status is recorded in status unless UPC_STATUS_IGNORE is passed in. 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.

6.4.5 The upc_waitall function

Synopsis

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

   void upc_waitall( int count,
                   upc_handle_t * handles,
                   upc_status_t * statuses );

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 and the associated index in the statuses array records the status unless UPC_STATUS_IGNORE is passed in.

6.4.6 The upc_testall function

Synopsis

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

   int upc_testall( int count,
                   upc_handle_t * handles,
                   upc_status_t * statuses );

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 and records the status of each unless UPC_STATUS_IGNORE is passed. Otherwise, a negative integer is returned. In this case, no handle is modified.

6.4.7 The upc_waitsome function

Synopsis

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

   int upc_waitsome( int count,
                    upc_handle_t * handles,
                    upc_status_t * statuses );

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 completes that operation as well as all others that are complete. The handles in the appropriate indexes are set to UPC_HANDLE_COMPLETE and the status information is set for corresponding index locations in the statuses array.

3. It returns a count of completed handles.
6.4.8 The upc_testsome function

Synopsis

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

   int upc_testsome( int count,
                     upc_handle_t * handles,
                     upc_status_t * statuses );`

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.

6.4.9 The upc_get_status function

Synopsis

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

   int upc_get_status( upc_handle_t handle,
                       upc_status_t * status );`

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. `status` is set as appropriate.

3. A subsequent call to `upc_wait` or `upc_test` is required to complete the operation.

References


   http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf.

http://www.gwu.edu/~upc/docs/UPC_Coll_Spec_V1.0.pdf.