# Proposal for Extending the UPC Memory Copy Library Functions Version 2.0 *

Dan Bonachea
University of California, Berkeley

March 22, 2007

bonachea@cs.berkeley.edu

## 1 Introduction

### 1.1 Abstract

This document outlines a proposal for extending UPC's point-to-point memcpy library with support for explicitly non-blocking transfers, and non-contiguous (indexed and strided) transfers. Various portions of this proposal could stand alone as independent extensions to the UPC library. The designs presented here are heavily influenced by analogous functionality which exists in other parallel communication systems, such as MPI, ARMCI, Titanium, and network hardware API's such as Quadrics elan, Infiniband vapi, IBM LAPI and Cray X-1.

Each section contains proposed extensions to the libraries in the UPC Language Specification (section 7).

### 1.2 Motivation

The UPC Language specification (version 1.1.1) provides a very minimal library for performing bulk-transfer communication. The *upc_memput*, *upc_memget* and *upc_memcpy* functions operate analogously to C99's *memcpy* function, and each provide the ability to move a single contiguous block of memory to/from locations specified by a pointer-to-local and pointer-to-shared or between locations specified by two pointers-to-shared. No further libraries are provided for directly expressing more complicated non-collective communication patterns - such as the movement of bulk data to/from non-contiguous locations (eg the column of a multi-dimensional array, or a set of locations in an irregular data structure). Non-contiguous access interfaces have historically been used to achieve speedups through communication aggregation - the transformation of fine-grained access patterns (which could naïvely be implemented using a large number of small messages), into more coarse-grained communication operations that improve network efficiency by sending larger messages and performing packing and unpacking at either end (possibly with hardware assistance). Furthermore, no mechanism is provided for the application programmer to express that a given communication operation can proceed independently with respect to other surrounding computation or communication operations

*The archival version of this document which should be cited in publication is: Bonachea, Dan. "Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v2.0", Lawrence Berkeley National Lab Tech Report LBNL-56495 v2.0

- a data independence property which has traditionally been used to obtain substantial parallel speedups by hiding communication latency with overlapped computation and other communication. The best an application programmer can currently do with the UPC 1.1.1 libraries is to express all non-collective bulk communication operations using a set of blocking contiguous transfers, and pray that very smart optimizers can transform this naïve access pattern (which has been forced by the restrictive library interface) into optimized communication operations that provide communication aggregation and overlap.

In an ideal world, UPC compilers could always automatically perform this transformation and achieve the maximal possible benefit from communication aggregation and overlap. However, the truth is that many factors force compilers to be overly conservative in such communication transformations and therefore the resulting communication pattern often falls quite short of the best one could hope to do with full application-level knowledge. Part of this failure is due to conservatism forced by features of UPC inherited from C (most notably pointer aliasing and separate compilation), however even assuming a perfect solution to these analysis problems, other fundamental sources of forced conservatism remain. In a complicated application, many important behavioral properties of the program are not directly expessed anywhere in the source program - they exist solely in the programmer's mind. Furthermore, many of the most useful properties (from an optimizer-writer's perspective) are not even possible to infer solely by inspection of the source program, even given an infinitely smart optimizer - because they depend on constraints such as the set of legal inputs, that are implicitly part of the program design but are often not expressed anywhere in the program, nor are they inferable solely from the program text. This is important because many of the most aggressive optimizations (such as some forms of static communication aggregation) need to make assumptions which are based on this sort of unexpressed knowledge in order to be safe (because for example, they might be unsafe in situations where the unexpressed assumptions are violated). Because the application programmer has this unexpressed algorithmic knowledge in his mind, he's in a unique position to direct these more aggressive optimizing transformations (given the proper tools), which even a perfect static compiler could not do without extra-linguistic help.

The UPC library extensions proposed in this document give the application programmer or library writer the tools necessary to request and express such beneficial transformations directly while tuning communication operations occuring in the application's critical path, rather than being constrained by the library interface to write communication in a naïve style and therefore being forced to rely upon a mythical perfect optimizer to automatically apply these important transformations (which we've just argued that the compiler often has insufficient information to legally perform). By allowing the programmer to directly and conveniently express communication aggregation and overlap in places where the algorithmic data dependencies allow, compiler implementors can focus their efforts on ensuring the requested communication is performed as efficiently as possible - for example leveraging available network hardware capabilities for non-blocking transfers and non-contiguous access.

## 1.3 Implementation Notes

All of the proposed extensions described in this document have been implementated and are available as a prototype implementation in the Berkeley UPC compiler, version 2.4.0 (http://upc.lbl.gov). All functions in the prototype implementation operate exactly as described in this document *with* the notable exception that all functions, types and constants named using the prefix *bupc_* instead of *upc_*. This naming convention reflects the fact that these extensions are not currently part of the official UPC language specification.

# 2 Explicit-handle non-blocking bulk-contiguous operations

The following functions provide non-blocking, split-phase memory access to shared data. All such non-blocking operations require an initiation (e.g., a put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed. These are "explicit-handle" non-blocking operations because the initiation function returns an explicit handle value, which must be passed to the synchronization function for completing the corresponding operation.

## 2.1 Non-blocking explicit handle type

```
type  upc_handle_t
value UPC_COMPLETE_HANDLE
```

The explicit-handle non-blocking data transfer functions return a *upc_handle_t* value to represent the non-blocking operation in flight. *upc_handle_t* is an opaque private data type whose contents are implementation-defined, with one exception - every implementation must provide a value corresponding to an "invalid" handle ($UPC\_COMPLETE\_HANDLE$) and furthermore this value must be the result of setting all the bits in the *upc_handle_t* data type to zero. Implementors are free to define the *upc_handle_t* type to be any reasonable and appropriate size, although they are recommended to use a type which fits within a single standard register on the target architecture. In any case, the data type should be wide enough to express at least $2^{16} - 1$ different handle values, to prevent limiting the number of non-blocking operations in progress due to the number of handles available.

It is legal for threads to pass *upc_handle_t* values into function callees or back to function callers. However, *upc_handle_t* values are thread-specific. In other words, it is an error to obtain a handle value by initiating a non-blocking operation on one thread, and later pass that handle value into a synchronization function from a different thread.

Any explicit-handle, non-blocking initiation operation may return the value $UPC\_COMPLETE\_HANDLE$ to indicate that the requested operation was completed synchronously. It is always an error to discard the *upc_handle_t* value for an explicit-handle operation in-flight - i.e. to initiate an operation and never synchronize on its completion.

## 2.2 Explicit-handle non-blocking operations

```
upc_handle_t upc_memcpy_async(shared void *dst, shared const void *src, size_t n);
upc_handle_t upc_memget_async(       void *dst, shared const void *src, size_t n);
upc_handle_t upc_memput_async(shared void *dst,        const void *src, size_t n);
upc_handle_t upc_memset_async(shared void *dst, int c, size_t n);
```

These operations have the same semantics as the corresponding functions defined in the UPC Language Specification section 7.2.5, except they are split-phase. The specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined.

## 2.3 Explicit-handle non-blocking synchronization

```
void upc_waitsync(upc_handle_t handle);
int upc_trysync(upc_handle_t handle);
```

Synchronize on the completion of a single specified explicit-handle non-blocking operation that was initiated by the calling thread. *upc_waitsync*() blocks until the specified operation has completed (or returns immediately if it has already completed). In any case, the handle value is "dead" after *upc_waitsync*() returns and may not be passed to future synchronization operations. *upc_trysync*() always returns immediately, with a non-zero value if the operation is complete (at which point the handle value is "dead", and may not be used in future synchronization operations), or zero if the operation is not yet complete and future synchronization is necessary to complete the corresponding operation. It is legal to pass $UPC\_COMPLETE\_HANDLE$ as input to these functions - *upc_waitsync* ($UPC\_COMPLETE\_HANDLE$) returns immediately and *upc_trysync* ($UPC\_COMPLETE\_HANDLE$) returns non-zero. It is an error to pass a *upc_handle_t* value (other than $UPC\_COMPLETE\_HANDLE$) for an operation which has already been successfully synchronized using one of the explicit-handle synchronization functions.

Note that the order in which non-blocking operations complete is intentionally unspecified - the system is free to coalesce and/or reorder non-blocking operations with respect to other blocking or non-blocking operations, or operations initiated from a separate thread - the only ordering constraints that must be satisfied are those explicitly enforced using the synchronization functions (i.e. the non-blocking operation is only guaranteed to occur somewhere in the interval between initiation and successful synchronization on that operation).

Implementors should attempt to make the non-blocking initiation operations return as quickly as possible - however in some cases (e.g. when a large number of non-blocking operations have been issued or the network is otherwise busy) it may be necessary to block temporarily while waiting for the network to become available. In any case, all implementations must support at least $2^{16}-1$ non-blocking operations in-progress per thread - that is, each thread is free to issue up to $2^{16}-1$ non-blocking operations before issuing a sync operation, and the implementation must handle this correctly without deadlock or livelock. Additionally, note that non-blocking operations proceed independently of barriers and other forms of inter-thread synchronization - these are not a substitute for *upc_waitsync*/*upc_trysync*.

**Example:** The following example demonstrates an explicitly asynchronous nearest neighbor exchange of data. We assume a regular domain decomposition in the data array A which is blocked in shared space. Each thread initiates a fetch of the neighbor data into local buffers, then performs independent computation while the communication proceeds overlapped in the background.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ];
double rightdata[BLKSZ];
upc_handle_t leftfetch_handle = UPC_COMPLETE_HANDLE;
upc_handle_t rightfetch_handle = UPC_COMPLETE_HANDLE;

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
  leftfetch_handle =  upc_memget_async(leftdata, &(A[BLKSZ*(MYTHREAD-1)]), BLKSZ*sizeof(double));
if (MYTHREAD < THREADS-1)  /* initiate fetch of data from right neighbor */
  rightfetch_handle = upc_memget_async(rightdata, &(A[BLKSZ*(MYTHREAD+1)]), BLKSZ*sizeof(double));

/* perform some independent computations here */

upc_waitsync(leftfetch_handle); /* block for completion of communication, if necessary */
upc_waitsync(rightfetch_handle);

/* now safe to operate on leftdata and rightdata */
```

## 2.4 Multiple explicit-handle non-blocking synchronization

The following convenience functions assist in synchronizing arrays of explicit handles:

```
void upc_waitsync_all (upc_handle_t *ph, size_t numhandles);
int  upc_trysync_all  (upc_handle_t *ph, size_t numhandles);
void upc_waitsync_some(upc_handle_t *ph, size_t numhandles);
int  upc_trysync_some (upc_handle_t *ph, size_t numhandles);
```

These functions synchronize on the completion of an array of explicit handles (all of which were created by the calling thread). *numhandles* specifies the number of handles in the provided array of handles. *upc_waitsync_all* blocks until all the specified operations have completed (or returns immediately if they have all already completed). *upc_trysync_all* always returns immediately, with a non-zero value if all the specified operations have completed, or a zero value if one or more of the operations is not yet complete and future synchronization is necessary to complete some of the operations. *upc_waitsync_some* blocks until at least one *incomplete* handle in the list has completed (where the incomplete handles are those which are not $UPC\_COMPLETE\_HANDLE$). *upc_trysync_some* always returns immediately, with a non-zero value if at least one incomplete handle in the provided array has completed, or a zero value if none of the incomplete handles in the provided array has completed.

All of these functions will modify the provided array to reflect completions - handles whose operations have completed are overwritten with the value $UPC\_COMPLETE\_HANDLE$, and the client may test against this value upon return to determine which operations are complete and which are still pending.

It is legal to pass the value $UPC\_COMPLETE\_HANDLE$ in some of the array entries, and the functions will ignore all such entries so that they have no effect on behavior. In the case where all entries in the array are $UPC\_COMPLETE\_HANDLE$ or $numhandles == 0$, then the wait variants will return immediately and the try variants will return immediately with a non-zero value to indicate success.

# 3 Implicit-handle non-blocking bulk-contiguous operations

The following functions provide non-blocking, split-phase access to shared data. All such non-blocking operations require an initiation (e.g., a put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed. These are "implicit-handle" non-blocking operations because the initiation function does not return a handle - rather, the operation becomes associated with an implicit handle owned by the calling thread, and all implicit-handle operations are synchronized together using a call to an implicit-handle synchronization function. These operations have the same (weak) ordering guarantees which apply to the explicit-handle variants.

## 3.1 Implicit-handle non-blocking operations

```
void upc_memcpy_asynci(shared void *dst, shared const void *src, size_t n);
void upc_memget_asynci(       void *dst, shared const void *src, size_t n);
void upc_memput_asynci(shared void *dst,        const void *src, size_t n);
void upc_memset_asynci(shared void *dst, int c, size_t n);
```

These operations have the same semantics as the corresponding functions defined in the UPC Language Specification section 7.2.5, except they are split-phase. The specified operation is initiated with a call to the above functions. The operation is not guaranteed to be complete until after the next successful call to *upc_waitsynci* or *upc_trysynci* made by the initiating thread (unless access region synchronization is in effect, as explained in section 4). The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined.

## 3.2 Implicit-handle non-blocking synchronization

The following functions are used to synchronize implicit-handle non-blocking operations:

```
void upc_waitsynci();
int upc_trysynci();
```

These functions synchronize the set of non-blocking implicit-handle operations previously issued by the calling thread outside any access region, and not yet synchronized through a successful implicit-handle synchronization. *upc_waitsynci* blocks until all operations in this set have completed (indicating these operations have been successfully synchronized). *upc_trysynci* tests whether all operations in the set have completed, and returns a non-zero value if so (which indicates these operations have been successfully synchronized) or zero otherwise (in which case **none** of these operations may be considered successfully synchronized).

If there are no outstanding implicit-handle operations (i.e., the set is empty), then *upc_waitsynci* returns immediately, and *upc_trysynci* returns immediately with a non-zero value to indicate success.

These functions notably do **not** synchronize any outstanding explicit-handle operations - those operations proceed independently and must be synchronized using the explicit-handle synchronization functions. Because the set of operations is determined dynamically and not lexically, implicit-handle synchronization functions can synchronize operations initiated within other function frames by the calling thread (but this cannot affect the correctness of correctly synchronized code - at worst it oversynchronizes).

**Example:** Here is the same example from section 2.3, written using implicit-handle synchronization. The example is semantically equivalent, but more concise as there are no explicit handles to manage.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ];
double rightdata[BLKSZ];

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
  upc_memget_asynci(leftdata, &(A[BLKSZ*(MYTHREAD-1)]), BLKSZ*sizeof(double));
if (MYTHREAD < THREADS-1)  /* initiate fetch of data from right neighbor */
  upc_memget_asynci(rightdata, &(A[BLKSZ*(MYTHREAD+1)]), BLKSZ*sizeof(double));

/* perform some independent computations here */

upc_waitsynci(); /* block for completion of communication, if necessary */

/* now safe to operate on leftdata and rightdata */
```

# 4 Access region synchronization

In some cases, it may be useful or desirable to initiate a number of non-blocking operations (possibly without knowing how many at compile-time) and synchronize them at a later time using a single, fast synchronization. Simple implicit handle synchronization may not be appropriate for this situation if there are intervening implicit accesses which are not to be synchronized. This situation could be handled using explicit-handle non-blocking operations and *upc_waitsync_all*, but this may not be desirable because it requires managing an array of handles (which may be inconvenient or costly when the number of operations is not known until runtime). To handle these cases, we provide *access region* synchronization, described below. It provides a useful middle ground between implicit and explicit handles in the expressiveness versus conciseness tradeoff.

## 4.1 Access region functions

```
void         upc_begin_accessregion();
upc_handle_t upc_end_accessregion();
```

The *upc_begin_accessregion* and *upc_end_accessregion* functions are used to define an access region - any statements which execute on the calling thread after a begin call and before the next end call are said to be *inside* the region. The begin and end calls must be paired, and may not be nested or the results are undefined. It is erroneous to call any implicit-handle synchronization function (section 3.2) inside an access region. All implicit-handle non-blocking operations initiated inside the region by the functions in section 3.1 become *associated* with the abstract access region handle being constructed. *upc_end_accessregion* returns an explicit handle which collectively represents all the associated operations (those implicit-handle operations initiated within the access region). This handle must later be passed to the regular explicit-handle synchronization functions in sections 2.3 and 2.4, and will be successfully synchronized when **all** of the associated operations initiated in the access region have completed. The associated operations are **not** synchronized by subsequent calls to the implicit-handle synchronization functions occurring after the access region (e.g. *upc_waitsynci*). Explicit-handle operations initiated within the access region operate as usual and do **not** become associated with the access region.

**Example:** Here is the same example from section 2.3, written using an access region. The example is semantically equivalent, but more concise as there is only one handle to manage.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS];
double leftdata[BLKSZ];
double rightdata[BLKSZ];

upc_begin_accessregion(); // begin the access region

if (MYTHREAD > 0) /* initiate fetch of data from left neighbor */
  upc_memget_asynci(leftdata, &(A[BLKSZ*(MYTHREAD-1)]), BLKSZ*sizeof(double));
if (MYTHREAD < THREADS-1)  /* initiate fetch of data from right neighbor */
  upc_memget_asynci(rightdata, &(A[BLKSZ*(MYTHREAD+1)]), BLKSZ*sizeof(double));

// end the access region and get the handle
upc_handle_t handle = upc_end_accessregion();

/* perform some independent computations here */

upc_waitsync(handle); /* block for completion of communication, if necessary */

/* now safe to operate on leftdata and rightdata */
```

**Example:** A more complicated example of an access region.

```
upc_begin_accessregion(); // begin the access region

upc_memput_asynci(...); // becomes associated with access region
while (...) {
  upc_memget_asynci(...); // becomes associated with access region
}

// unrelated explicit-handle operation not associated with access region
upc_handle_t h2 = upc_memget_async(...);
upc_waitsync(h2);

// end the access region and get the handle
upc_handle_t handle = upc_end_accessregion();

.... // other code, which may include unrelated implicit or explicit handle
     // operations+syncs, or other access regions, etc

// wait for all the operations associated with the access region to complete
upc_waitsync(handle);
```

# 5 Indexed/Vector memcpy operations

The indexed memcpy functions provide a general mechanism to express an operation which gathers data from arbitrary source regions of memory and scatters data into arbitrary destination regions of memory. Expressing such a data movement pattern as a single high-level operation (as opposed to many small, contiguous operations) allows for more aggressive optimization of the data movement within the UPC implementation - for example, tuning the transfer mechanism for maximal performance on the given memory hierarchy or taking advantage of platform-specific scatter/gather support in network hardware. All the functions are non-collective - they are called by a single thread to initiate an indexed memory copy transfer.

## 5.1 Common Requirements

The total amount of data specified by the source regions must equal the total amount of data specified by the destination regions (although the individual regions in each list need not be of equal size). In other words, counts and lengths in the source and destination lists need not match, so long as they both specify the same total amount of data. The effect of the operation is that data is copied from the source regions, in the order specified by *srclist*, to the destination regions, in the order specified by *dstlist*. Note the contents of the destination regions is undefined while the operation is in progress (i.e. the actual order in which the writes take place is undefined), and if the contents of the source regions change while the operation is in progress the result is undefined.

The destination regions must be completely disjoint and must not overlap with any source regions, otherwise the result is undefined. Source regions are permitted to overlap with each other.

If *dstcount* and *srccount* are zero, the operation is a no-op and the other arguments are ignored.

## 5.2 Possible Design A - List of variable-sized regions

```
typedef struct {
  void *addr;
  size_t len;
} upc_pmemvec_t;

typedef struct {
  shared void *addr; // treated as a (shared [] char *) - ie. no wrapping
  size_t len;
} upc_smemvec_t;
```

A *upc_pmemvec_t* specifies a contiguous region of local memory valid on the current thread starting at base address *addr* and extending for *len* bytes. A *upc_smemvec_t* specifies a contiguous region of shared memory with affinity to a single thread, starting at base address *addr* and extending for *len* bytes. In both cases *len* may be zero, in which case that entry is ignored.

```
void upc_memcpy_vlist(size_t dstcount, upc_smemvec_t const dstlist[],
                      size_t srccount, upc_smemvec_t const srclist[]);
void upc_memput_vlist(size_t dstcount, upc_smemvec_t const dstlist[],
                      size_t srccount, upc_pmemvec_t const srclist[]);
void upc_memget_vlist(size_t dstcount, upc_pmemvec_t const dstlist[],
                      size_t srccount, upc_smemvec_t const srclist[]);
```

```
upc_handle_t upc_memcpy_vlist_async(size_t dstcount, upc_smemvec_t const dstlist[],
                                    size_t srccount, upc_smemvec_t const srclist[]);
upc_handle_t upc_memput_vlist_async(size_t dstcount, upc_smemvec_t const dstlist[],
                                    size_t srccount, upc_pmemvec_t const srclist[]);
upc_handle_t upc_memget_vlist_async(size_t dstcount, upc_pmemvec_t const dstlist[],
                                    size_t srccount, upc_smemvec_t const srclist[]);
```

- *srclist* and *dstlist* specify a list of contiguous memory regions to be used as the source and destination for the memory transfer. Each *upc_smemvec_t* entry is permitted to specify data with affinity to a different thread.
- *srccount* and *dstcount* indicate the number of region entries in the *srclist* and *dstlist* array, respectively.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined. The *srclist* and *dstlist* arrays must remain valid and unchanged until the operation is complete.

**Dan's Comments**
**PROS:** good for specifying bounding boxes, efficiently allows packing in a contiguous buffer at either end, allows multiple remote affinities, mirrors the UPC-IO List IO interface
**CONS:** bad for vectorization, high metadata space consumption, full generality provided may not map well to more restrictive lower-level scatter/gather network layers

**Example:** The following example demonstrates the use of *upc_memget_vlist* (Design A) to fetch some individual elements, a group of elements, and an entire block in a single operation into a single, contiguous local buffer. For demonstration purposes the data was fetched from shared memory with affinity to different threads, although this need not always be the case.

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS]; /* assume THREADS >= 3 */
upc_smemvec_t srclist[] = {
        { &(A[14]), sizeof(double) },     /* element 14 (from thread 0) */
        { &(A[20]), sizeof(double) },     /* element 20 (from thread 0) */
        { &(A[100]), 50*sizeof(double) }, /* elements 100..149 (from thread 1) */
        { &(A[2*BLKSZ]), BLKSZ*sizeof(double) } /* entire block (from thread 2) */
      };
double mybuf[52+BLKSZ];
upc_pmemvec_t dstlist[] = { { mybuf, sizeof(mybuf) } };

upc_memget_vlist(1, dstlist, 4, srclist);

/* compute on contents of mybuf */
```

### 5.3 Possible Design B - List of fixed-size regions

```
void upc_memcpy_ilist(size_t dstcount, shared       void * const dstlist[], size_t dstlen,
                      size_t srccount, shared const void * const srclist[], size_t srclen);
void upc_memput_ilist(size_t dstcount, shared       void * const dstlist[], size_t dstlen,
                      size_t srccount,        const void * const srclist[], size_t srclen);
void upc_memget_ilist(size_t dstcount,              void * const dstlist[], size_t dstlen,
                      size_t srccount, shared const void * const srclist[], size_t srclen);

upc_handle_t upc_memcpy_ilist_async(size_t dstcount, shared       void * const dstlist[],
                                    size_t dstlen,
                                    size_t srccount, shared const void * const srclist[],
                                    size_t srclen);
upc_handle_t upc_memput_ilist_async(size_t dstcount, shared       void * const dstlist[],
                                    size_t dstlen,
                                    size_t srccount,        const void * const srclist[],
                                    size_t srclen);
upc_handle_t upc_memget_ilist_async(size_t dstcount,              void * const dstlist[],
                                    size_t dstlen,
                                    size_t srccount, shared const void * const srclist[],
                                    size_t srclen);
```

These functions copy data elements as *srccount* contiguous regions of memory with fixed length *srclen* from base addresses $srclist[0]...srclist[srccount-1]$, and place the data as into contiguous regions of memory with length *dstlen* at base addresses $dstlist[0]...dstlist[dstcount-1]$.

- *srclist* and *dstlist* specify a list of element addresses be used as the source and destination for the memory transfer. Each entry is permitted to specify data with affinity to a different thread.
- *srccount* and *dstcount* indicate the number of elements in the *srclist* and *dstlist* array, respectively.
- *srclen* and *dstlen* specify the length in bytes for each contiguous region referenced by *srclist* and *dstlist*. The two need not be equal, but must both be greater than zero.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined. The *srclist* and *dstlist* arrays must remain valid and unchanged until the operation is complete.

**Dan's Comments**
**PROS:** minimizes metadata space overhead, allows multiple remote affinities, efficiently allows packing in a contiguous buffer at either end
**CONS:** can't efficiently handle different-sized regions in a single operation, some platforms may perform badly when *srclen* and *dstlen* are unequal.

**Example:** The following example demonstrates the use of *upc_memget_ilist* (Design B) to fetch some individual elements into a single, contiguous local buffer. For demonstration purposes the data was fetched from shared memory with affinity to different threads, although this need not always be the case. Note that each region of source memory in a single operation is constrained to be the same size (although it needn't match the underlying element size). The most concise way to fetch many regions of different sizes with this interface is to use a separate operation for each region size (and possibly use asynchronous operations to improve concurrency).

```
#define BLKSZ 100
shared [BLKSZ] double A[BLKSZ*THREADS]; /* assume THREADS >= 2 */
shared void * srclist[] = {
        &(A[14]), &(A[15]), &(A[16]),   /* element 14..16 (from thread 0) */
        &(A[100]), &(A[110])            /* element 100 and 110 (from thread 1) */
      };
double mybuf[5];
void * dstlist[] = { &mybuf };

upc_memget_ilist(1, dstlist, 5*sizeof(double),
                5, srclist, sizeof(double));

/* compute on contents of mybuf */
```

# 6 Strided memcpy

The strided memcpy functions are a special case of the indexed memcpy functions, with an interface specialized for efficiently expressing copies of arbitrary rectangular sections of dense multi-dimensional arrays. All the functions are non-collective - they are called by a single thread to initiate a strided memory copy transfer.

## 6.1 Possible Design A - fixed region size/stride (2-d rectangular array section)

This design option has a relatively simple but restrictive interface - operating on fixed size regions (chunks), with a single fixed stride through linear memory between each chunk.

```
void upc_memcpy_fstrided(shared void *dstaddr,  size_t dstchunklen,
                         size_t dstchunkstride, size_t dstchunkcount,
                         shared void *srcaddr,  size_t srcchunklen,
                         size_t srcchunkstride, size_t srcchunkcount);
void upc_memput_fstrided(shared void *dstaddr,  size_t dstchunklen,
                         size_t dstchunkstride, size_t dstchunkcount,
                                void *srcaddr,  size_t srcchunklen,
                         size_t srcchunkstride, size_t srcchunkcount);
void upc_memget_fstrided(       void *dstaddr,  size_t dstchunklen,
                         size_t dstchunkstride, size_t dstchunkcount,
                         shared void *srcaddr,  size_t srcchunklen,
                         size_t srcchunkstride, size_t srcchunkcount);

upc_handle_t upc_memcpy_fstrided_async(shared void *dstaddr,  size_t dstchunklen,
                                       size_t dstchunkstride, size_t dstchunkcount,
                                       shared void *srcaddr,  size_t srcchunklen,
                                       size_t srcchunkstride, size_t srcchunkcount);
upc_handle_t upc_memput_fstrided_async(shared void *dstaddr,  size_t dstchunklen,
                                       size_t dstchunkstride, size_t dstchunkcount,
                                              void *srcaddr,  size_t srcchunklen,
                                       size_t srcchunkstride, size_t srcchunkcount);
upc_handle_t upc_memget_fstrided_async(       void *dstaddr,  size_t dstchunklen,
                                       size_t dstchunkstride, size_t dstchunkcount,
                                       shared void *srcaddr,  size_t srcchunklen,
                                       size_t srcchunkstride, size_t srcchunkcount);
```

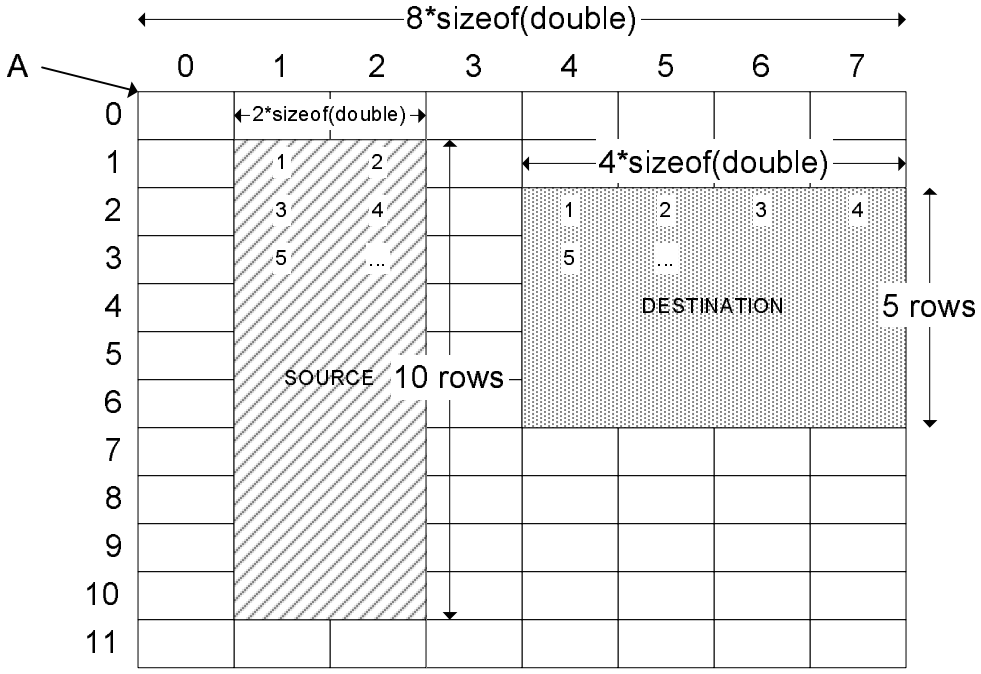
- *srcaddr* and *dstaddr* base addresses for the source and destination regions, treated as a (shared [] char *) - ie. no wrapping
- *srcchunklen* and *dstchunklen* length of each chunk in bytes
- *srcchunkstride* and *dstchunkstride* number of bytes between the start of each chunk (must be >= *chunklen*)
- *srcchunkcount* and *dstchunkcount* number of chunks

The total data length in the source and destination must be equal, i.e., $srcchunklen * srcchunkcount == dstchunklen * dstchunkcount$. If the source locations overlap any destination locations, the result is unde-

fined. If $srcchunklen * srcchunkcount$ and $dstchunklen * dstchunkcount$ are zero, the operation is a no-op and the other arguments are ignored.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined.

**Dan's Comments**
**PROS:** simplicity of interface arguments, regular sparse access, efficiently allows packing in a contiguous buffer at either end, compact metadata, allows the copied region and underlying arrays to differ in shape at the source and destination
**CONS:** lacks generality - unable to retrieve an arbitrary rectangular array section in more than two dimensions




```
shared [] double A[12][8];
...
upc_memcpy_fstrided(&(A[2][4]), 4*sizeof(double), 8*sizeof(double), 5,
                                &(A[1][1]), 2*sizeof(double), 8*sizeof(double), 10);
```

Figure 1: Example of upc_memcpy_fstrided, Design A

## 6.2 Possible Design B - N-d rectangular array section

```
void upc_memcpy_strided(shared       void *dstaddr, const size_t dststrides[],
                        shared const void *srcaddr, const size_t srcstrides[],
                        const size_t count[], size_t stridelevels);
void upc_memput_strided(shared       void *dstaddr, const size_t dststrides[],
                               const void *srcaddr, const size_t srcstrides[],
                        const size_t count[], size_t stridelevels);
void upc_memget_strided(             void *dstaddr, const size_t dststrides[],
                        shared const void *srcaddr, const size_t srcstrides[],
                        const size_t count[], size_t stridelevels);

upc_handle_t upc_memcpy_strided_async(shared       void *dstaddr, const size_t dststrides[],
                                      shared const void *srcaddr, const size_t srcstrides[],
                                      const size_t count[], size_t stridelevels);
upc_handle_t upc_memput_strided_async(shared       void *dstaddr, const size_t dststrides[],
                                             const void *srcaddr, const size_t srcstrides[],
                                      const size_t count[], size_t stridelevels);
upc_handle_t upc_memget_strided_async(             void *dstaddr, const size_t dststrides[],
                                      shared const void *srcaddr, const size_t srcstrides[],
                                             const size_t count[], size_t stridelevels);
```

- *srcaddr* Source starting address of the data block to copy, treated as a (shared [] char *) (i.e., no wrapping).
- *srcstrides* Source array of positive stride distances in bytes to move along each dimension. (*stridelevels* entries)
- *dstaddr* Destination starting address of the data block to receive the copy, treated as a (shared [] char *) (i.e., no wrapping).
- *dststrides* Destination array of positive stride distances in bytes to move along each dimension. (*stridelevels* entries)
- *count* Slice size in each dimension. *count*[0] should be the number of bytes of contiguous data in the leading (rightmost) dimension. (*stridelevels* + 1 entries)
- *stridelevels* The level of strides (for an N-d array copy, one generally sets *stridelevels* == $(N-1)$).

If the source locations overlap any destination locations, the result is undefined. If *stridelevels* is zero, the operation is a contiguous copy of *count*[0] bytes, and the *srcstrides* and *dststrides* arguments are ignored. If any entry in *count*[0..*stridelevels*] is zero, the operation is a no-op and the other arguments are ignored. The dimensional strides in *srcstrides* and *dststrides* must be monotonically increasing and must not specify overlapping locations - more specifically, $srcstrides[0] \geq count[0] \ \wedge \ \forall i \in [1..(stridelevels - 1)] \mid srcstrides[i] \geq (count[i] * srcstrides[i-1])$, and accordingly for *dststrides*.

For the async variants, the specified operation is initiated with a call to the above functions which return an explicit handle representing the operation in-flight. The operation is not guaranteed to be complete until after a successful call to *upc_waitsync* or *upc_trysync* on the returned handle. The contents of all affected destination memory is undefined while the operation is in-flight, and if the contents of any source memory changes while the operation is in-flight, the result is undefined. The *srcstrides*, *dststrides*, and *count* arrays must remain valid and unchanged until the operation is complete.

**Dan's Comments**
**PROS:** fully general - can take an arbitrary rectangular section from a dense rectangular array of any dimensionality, efficiently allows packing in a contiguous buffer at either end, allows the underlying arrays to differ in shape at the source and destination
**CONS:** interface complexity may intimidate novice users, does not allow the copied region to differ in shape at the source and destination (i.e., the rectangular section being copied must have the same extents in N-d space at either end)

**Example:** To put a 3-d block of data, shaped 2x3x4, starting at location (5, 6, 7) in A to B in location (8, 9, 10), the arguments to *upc_memput_strided* can be set as follows:

```
double A[11][12][13]; /* local array */
shared [] double B[14][15][16]; /* remote array */

void * srcaddr;
shared void * dstaddr;
size_t count[3];
size_t stridelevels;

srcaddr = &(A[5][6][7]);
srcstrides[0] = 13 * sizeof(double);      /* stride in bytes for the rightmost dimension */
srcstrides[1] = 12 * 13 * sizeof(double); /* stride in bytes for the middle dimension */
dstaddr = &(B[8][9][10]);
dststrides[0] = 16 * sizeof(double);      /* stride in bytes for the rightmost dimension */
dststrides[1] = 15 * 16 * sizeof(double); /* stride in bytes for the middle dimension */
count[0] = 4 * sizeof(double); /* number of bytes of contiguous data (width in rightmost dimension) */
count[1] = 3; /* width in middle dimension */
count[2] = 2; /* width in leftmost dimension */
stridelevels = 2;

upc_memput_strided(srcaddr, dststrides, dstaddr, srcstrides, count, stridelevels);
```

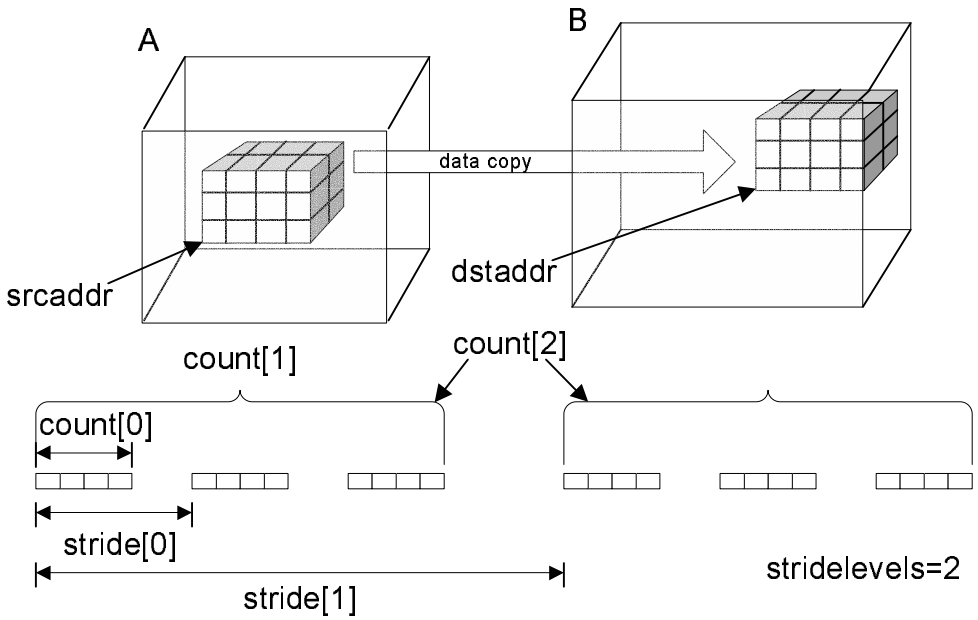



Figure 2: Illustration of a 3-d *upc_memput_strided* (Design B), and the in-memory data layout of the source or destination

# 7 Appendix: Open Issues and Possible Extensions

1. **Non-collective reblocking shared data movement**
   Consider providing non-collective memcpy mechanisms that directly support operating over a distributed array.

   Nothing in UPC currently provides a way to directly express non-collective automatic reblocking of arrays (i.e., allow a single thread to request shuffling of data to change the effective blocking factor of an array, especially for gathering to/from an indefinitely blocked array), although this seems like something we should eventually explore.

   Note that although it's not entirely elegant, one certainly can use the proposed scatter/gather functions to do the required communication in a single operation, ie:

```
/* assuming indexed memcpy design A */
shared [BLKSZ] double A[NUMELEM];
double localA[NUMELEM];
upc_pmemvec_t dst = { &localA, NUMELEM*sizeof(double) };
upc_smemvec_t myvec[THREADS];
for (int i=0; i < THREADS; i++) {
  myvec[i].addr = &A[BLKSZ*i];
  myvec[i].len = upc_affinitysize(NUMELEM*sizeof(double),
                                  BLKSZ*sizeof(double), i);
}
upc_memget_vlist(1, dst, THREADS, myvec);
```

   The code above gathers the pieces of the $A$ array with affinity to each thread into a single, private contiguous buffer using a single operation (and orders them in the buffer by former thread affinity). If $NUMELEM > BLKSZ * THREADS$ (i.e., the blocks wrap around back to thread 0) and we want the data ordered by block number, we can use a slightly longer loop, that should still perform well for reasonably large block sizes:

```
/* assuming indexed memcpy design A */
shared [BLKSZ] double A[NUMELEM];
double localA[NUMELEM];
upc_pmemvec_t dst = { &localA, NUMELEM*sizeof(double) };
upc_smemvec_t myvec[NUMELEM/BLKSZ + 1];
shared [BLKSZ] double *p = A;
for (int i=0; i < NUMELEM/BLKSZ; i++) {
  myvec[i].addr = p;
  myvec[i].len = BLKSZ*sizeof(double);
  p += BLKSZ;
}
int leftoverelems = (&A[NUMELEM]-p);
if (leftoverelems > 0) {
  myvec[i].addr = p;
  myvec[i].len = leftoverelems*sizeof(double);
  i++;
}
upc_memget_vlist(1, dst, i, myvec);
```

   Note the same approaches also easily work under indexed memcpy design B (fixed-width regions) when $NUMELEMS\%BLKSZ == 0$ (and otherwise can be made to work with one additional separate memget of the left-over elements in the final partial block).

2. **Consider supporting strided source/destination that spans affinities**
   Currently the entire source region of a strided operation must have affinity to a single thread (and similarly for the destination region). If we ever add direct support for non-collective reblocking data movement, we might also consider extending the strided operations to work over distributed arrays (i.e., take a blocksize parameter as input). However, the strided interface is already quite high on the complexity scale, and this extension may scare off additional users. Furthermore, adding a blocksize parameter to the strided interface significantly complicates the pointer arithmetic in the implementation of the general block-distributed case, reducing performance (at least for that case) and increasing the testing/development burden of implementation.

3. **Consider allowing reshaping N-d strided transfers**
   The N-d strided interface (i.e., design B) does not allow the copied region to differ in shape at the source and destination (i.e., the rectangular section being copied must have the same extents in N-d space at either end). Note the interface *does* permit the underlying N-d arrays to differ in their dimensions, and it *does* efficiently allow transfers to/from a contiguous buffer at either end. However, it does not allow one to take the elements from an arbitrary N-d rectangular section at the source and shuffle them into an arbitrary N-d rectangular section of different shape (and equal volume) at the destination. The interface could be adapted to support this (bizarre?) usage by splitting the *count* array into *srccount* and *dstcount* (adding to the complexity of the interface and implementation) but it was perceived that there was no demand for the additional generality.

4. **Remote completion (target notification)**
   In some algorithms, one may want the ability to initiate a point-to-point non-blocking operation and allow the target thread (rather than the initiator) to synchronize on the completion of the operation. However, it's unclear how such an interface would look for UPC or even if it's consistent with UPC's general philosophy of one-sided communication through globally shared memory with logical affinity (since such a primitive is really just send/recv two-sided message passing in disguise - the only significant difference being that the initiator provides all the relevant memory addresses).

5. **Explicitly non-blocking UPC collectives and IO**
   All the collective and IO functions could be enhanced with handle-based non-blocking versions. Because these functions are collective, this should be done with a *different* collective handle type (e.g., *upc_all_handle_t*) and corresponding collective synchronizations functions (e.g., *upc_all_waitsync* / *upc_all_trysync*).

6. **Consider relaxing the required lifetime of the input metadata arrays**
   Currently the async UPC functions that take metadata input (e.g., address lists) in array form require those metadata arrays to remain unchanged until the operation has been successfully synchronized. This decision was motivated by the desire to provide the greatest freedom to implementors - this guarantee may allow an implementation to avoid copying the metadata inputs, and therefore provide better performance. The user is already required to ensure the source data remains unchanged while the operation is in progress (again, to avoid requiring synchronous copying overhead in the async initiation functions), so it doesn't seem overly burdensome to additionally require the metadata arrays to remain unchanged until the async operation has been synchronized. However, if this becomes problematic for applications in practice, then we could consider relaxing the lifetime requirement for the metadata arrays.

7. **Clean up the limit on the number of outstanding async operations**
   We basically want a limit which is guaranteed to be high enough such that application writers and code generators never have to worry about it (ie firmly disallow implementations that provide some paltry amount, like four non-blocking operations), but clearly an unbounded number of outstanding operations is not efficiently supportable (due to handle representation constraints, if nothing else).

   We may want to provide a compile-time constant defined by the implementation (e.g., $UPC\_MAX\_ASYNC\_INFLIGHT$) that specifies a per-thread limit on how many async operations are permitted

to be in-flight (unsynced) at any given time, and furthermore require all implementations to provide a value $UPC\_MAX\_ASYNC\_INFLIGHT >= 2^{16} - 1$.

8. **Provide a transpose operation**
   The monotonicity restrictions on the contents of *srcstrides*/*dststrides* in the strided API (design B) effectively imply that one cannot transpose the dimensions of the copied region during a strided copy using this interface. Given that transpose is a frequently-used operation (and one that may need to be done to/from remote memory), it may be worthwhile to add a version of the strided interface which allows one to specify a transpositional strided copy. This should be a separate function for documentation reasons (it's conceptually different than copy) and because efficient implementations are likely to differ considerably from the non-transpositional case. In addition to relaxing the monotonicity property, we'd also want to add one more element to the *srcstrides*/*dststrides* array so the user can explicitly indicate the dimension with unit-stride contiguity (in the current interface, the lowest order dimension always has an implicit stride of 1, which is not true in a transpositional copy). If we choose to provide this extension, we may additionally consider allowing negative stride values, which would cause the transposition to execute a negative injection on the index space (i.e., values would be "reflected" across a dimension, effectively "flipping over" the values in the rows along the given direction).

9. **Provide wrappers for 2-d and 3-d strided copy**
   Given that 2-d and 3-d arrays are so commonly used, we could provide wrapper functions around the strided copy (design B) function which take all the necessary parameters as values and construct the metadata expected by the general N-d strided copy function. There would be some overhead associated with the wrapper (especially if the metadata lifetime requirement remains unchanged), but it would provide a simpler interface for less sophisticated users.