Proposal for Extending the UPC Libraries with Explicit
Point-to-Point Synchronization Support
Version 0.1 *

Dan Bonachea
University of California, Berkeley

May 16, 2005

bonachea@cs.berkeley.edu

1 Introduction

1.1 Abstract
This document outlines a proposal for extending the Unified Parallel C (UPC) library with support for explicit point-to-point synchronization and synchronizing memory copies. The interface design is partly based on the POSIX semaphore interface, with extensions appropriate to the UPC global address space memory model.

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

1.2 Motivation
The UPC Language specification (version 1.2) provides a sophisticated hybrid memory consistency model, which allows users to select on a per-access basis whether memory operations issued to shared memory follow a sequentially-consistent (strict) memory model, or a more relaxed model akin to release consistency which allows for greater optimization possibilities and thereby enables higher performance. This set of primitive shared memory operations is sufficient to implement arbitrary inter-thread synchronization using standard shared-memory synchronization algorithms, however constructing such synchronization correctly and efficiently can often be subtle and error-prone, especially for novice users unfamiliar with the nuances of the UPC memory consistency model. The UPC library already provides a shared lock abstraction (upc_lock_t) that can be used to conveniently construct critical sections and enforce mutual exclusion between threads – however the library does not currently include functions designed to efficiently support producer-consumer synchronization patterns. Specifically lacking is library functions enabling threads to perform efficient pairwise synchronization, a capability which is often required when exchanging shared data with neighboring threads in loosely-synchronized applications. The library extensions proposed in this document provide a commonly-needed set of point-to-point synchronization constructs which have proven useful in a number of UPC applications.

*This document is a draft - please do not cite in publication.
It’s important to note that all of the functionality described in this proposal can be implemented directly in UPC with appropriate use of strict/relaxed operations and carefully-coded spin-waiting – this interface does not add any fundamental new functionality, and therefore is entirely amenable to reference implementations written directly in UPC. However, the proposed interface exposes this functionality in a convenient abstraction that reduces the programmer effort required to correctly and efficiently implement point-to-point synchronization. In some cases (notably for loosely coupled cluster systems) the implementation of certain functions in this library might additionally perform more slightly efficiently than an equivalent program expressed directly in UPC.

1.3 Implementation Notes

All of the proposed extensions described in this document have been implemented and are available as a prototype implementation in the Berkeley UPC compiler, version 2.XXX (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 Point-to-Point Synchronization using Semaphores

The following library functions provide a semaphore abstraction that can be used to perform efficient and convenient point-to-point synchronization between the threads of a UPC program.

2.1 Common Requirements

All the functions described in this document non-collective - they are called independently by individual threads to initiate or complete point-to-point synchronization operations (however, in the course of such synchronization threads will sometimes block awaiting actions by other threads, as required by the synchronization semantics).

2.2 Semaphore type

Synopsis:

#include <upc.h>

type upc_sem_t

Description:

All of the point-to-point synchronization operations in this library operate on a upc_sem_t object, which logically represents a semaphore (akin to POSIX semaphores) which resides in shared memory with affinity to a particular thread. Fundamentally, semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.

The type upc_sem_t is an opaque UPC type. upc_sem_t is a shared datatype with incomplete type (as defined in [ISO/IEC00 Sec 6.2.5]). Objects of type upc_sem_t may therefore only be manipulated through pointers.
Such objects represent an atomic counter, and as such their logical state consists of an integral, non-negative value which is the current value of the counter. Two pointers to that reference the same semaphore object will compare as equal. The results of applying \texttt{upc\_phaseof()} and \texttt{upc\_addrfield()} to such pointers are undefined, however \texttt{upc\_threadof()} may be used to query the thread affinity of a \texttt{upc\_sem\_t} object, which always corresponds to the thread which created the object (via a prior call to \texttt{upc\_sem\_alloc}).

### 2.3 Semaphore Allocation and Destruction

**Synopsis:**

```c
#include <upc.h>

upc_sem_t *upc_sem_alloc(int flags);
void upc_sem_free(upc_sem_t *s);
```

**Description:**

\texttt{upc\_sem\_alloc} non-collectively allocates a semaphore with affinity to the calling thread, initialized to the logical value zero. The \texttt{flags} argument indicates the variety of semaphore to be created, which constrains subsequent usage of the object (and possibly allows for greater optimization potential for the more constrained usage models). The flags argument should be a logical or of the following integral constants, defined in \texttt{upc.h}:

\begin{verbatim}
UPC\_SEM\_BOOLEAN
UPC\_SEM\_INTEGER
UPC\_SEM\_SPRODUCER
UPC\_SEM\_MPRODUCER
UPC\_SEM\_SCONSUMER
UPC\_SEM\_MCONSUMER
\end{verbatim}

\texttt{UPC\_SEM\_BOOLEAN} / \texttt{UPC\_SEM\_INTEGER} are mutually exclusive, as are \texttt{UPC\_SEM\_SPRODUCER} / \texttt{UPC\_SEM\_MPRODUCER} and \texttt{UPC\_SEM\_SCONSUMER} / \texttt{UPC\_SEM\_MCONSUMER}.

\texttt{UPC\_SEM\_BOOLEAN} indicates the semaphore will only ever contain the values zero and one. Incrementing a boolean semaphore which already has value one has no effect. It is an error to attempt to decrement a boolean semaphore by more than one. \texttt{UPC\_SEM\_INTEGER} indicates a fully general integer semaphore which may contain any non-negative value less than \texttt{UPC\_SEM\_MAXVALUE}, which is guaranteed to be at least 65535.

\texttt{UPC\_SEM\_SPRODUCER} indicates the semaphore may only be incremented by one specific thread over the entire lifetime of the object, and behavior is undefined otherwise. \texttt{UPC\_SEM\_MPRODUCER} allows increments from any thread.

\texttt{UPC\_SEM\_SCONSUMER} indicates the semaphore may only be decremented by the current thread (the one creating the object who will share its affinity) and behavior is undefined otherwise. \texttt{UPC\_SEM\_MCONSUMER} indicates a semaphore that may be decremented by any thread (NOT YET IMPLEMENTED).

Passing \texttt{flags} == 0 selects most general (safest) configuration:

\texttt{UPC\_SEM\_INTEGER}\texttt{UPC\_SEM\_MPRODUCER}\texttt{UPC\_SEM\_MCONSUMER}

\texttt{upc\_sem\_free} may be called by any single thread to release the resources associated with the given semaphore. Behavior of subsequent operations on the given semaphore are undefined, and if any thread is concurrently accessing the semaphore (e.g., blocked inside \texttt{upc\_sem\_wait}) the behavior is also undefined.

3
2.4 Point-to-Point Synchronization Operations

Synopsis:

```c
#include <upc.h>

void upc_sem_post(upc_sem_t *s);
void upc_sem_postN(upc_sem_t *s, size_t n); /* only valid for INTEGER sems */

void upc_sem_wait(upc_sem_t *s);
void upc_sem_waitN(upc_sem_t *s, size_t n); /* only valid for INTEGER sems */

int upc_sem_try(upc_sem_t *s);
int upc_sem_tryN(upc_sem_t *s, size_t n); /* only valid for INTEGER sems */
```

Description:

`upc_sem_post(N)`: atomically increment the logical value of semaphore `s` by 1 (`N`)

`upc_sem_wait(N)`: suspend the calling thread until the logical value of semaphore `s` is \( i = 1 \) (`N`), then atomically decrement the value by that amount and return. If multiple threads are simultaneously blocked inside `wait`, (only valid for `UPC_SEM_MCONSUMER`) then it is undefined the order in which they will be serviced (no fairness guarantees)

`upc_sem_try(N)`: A non-blocking variant of `upc_sem_wait(N)`. Attempt to perform a `upc_sem_wait(N)` on `s`. If the operation can succeed immediately, perform it and return non-zero to indicate success. Otherwise, return zero to indicate failure.

`upc_sem_post(N)` implies a `upc_fence` operation upon entry to the function, `upc_sem_wait(N)` implies a `upc_fence` operation immediately before exiting, and `upc_sem_try(N)` implies a `upc_fence` operation immediately before a successful completion.
Example: The following example demonstrates ...

```c
#include <upc.h>

bupc_sem_t * shared allflags[THREADS];
shared int applicationData[THREADS];

int main() {

    /* create semaphores and exchange information */
    upc_sem_t *myflag = upc_sem_alloc(0);
    allflags[MYTHREAD] = myflag;
    upc_barrier;
    int partnerid = MYTHREAD^1;
    upc_sem_t *partnerflag = allflags[partnerid]; /* fetch ptr from remote */

    /* use the semaphores to perform producer-consumer sync */
    if (!MYTHREAD&1) { /* even threads */
        /* produce some result for my partner thread */
        applicationData[partnerid] = computeSomethingInteresting();

        upc_sem_post(partnerflag); /* send notification to my partner */
    } else { /* odd threads */

        upc_sem_wait(myflag); /* wait for notification from partner */

        /* consume data produced by my partner thread */
        computeSomethingElse(applicationData[MYTHREAD]);
    }
}
```
3 Signalling Data Movement Operations

High-level description...

3.1 Signalling upc_memput

Synopsis:

```
#include <upc.h>

void upc_memput_signal (shared void *dst, const void *src, size_t nbytes,
                     upc_sem_t *s, size_t n);
void upc_memput_signal_async(shared void *dst, const void *src, size_t nbytes,
                     upc_sem_t *s, size_t n);
```

Description:

Perform a memput with the same data movement semantics as `upc_memput`, and increment `s` by `n` when the transfer is complete. Requires `upc_threadof(s) == upc_threadof(dst)`.

Both functions `MAY` return on the initiator before the transfer is complete at the target – the semaphore on the target will be atomically incremented by `n` when the transfer is globally complete.

No explicit notifications or guarantees are provided to the initiator regarding the completion of the transfer at the target (remote completion).

`bupc_memput_signal` returns as soon as the source memory is safe to overwrite (i.e., it blocks for local completion of the transfer), whereas `bupc_memput_signal_async` `MAY` return earlier, while the source memory is still in use (and therefore not safe to overwrite). Callers of `bupc_memput_signal_async` are responsible for enforcing their own synchronization from the target thread to the initiator thread, in order to decide when the source memory is safe to overwrite.
Appendix: Open Issues and Possible Extensions

1. Remote wait
   There’s a semantic question of whether we should allow threads to `upc_wait(N)` on `upc_sem_t` objects with remote affinity. This should never be necessary in producer-consumer data exchanges where the consumer is known to the producer (and thus the data and signal can be pushed to the consumer), however it might be useful in multiple-consumer situations where there may be several possible consumers, and the one to consume a particular signal may depend on dynamic system behavior in the produce-consume interval.

   The capability to `upc_wait(N)` on a remote `upc_sem_t` unfortunately introduces some implementation complexities that may negatively impact the performance of all cases (which is why it is currently prohibited).

2. Allocation Flags
   Are all the allocation flags useful? Is additional information desirable?

3. Other misc POSIX-like functionality
   Missing POSIX features that may turn out to be useful:
   - Something like `sem_getvalue`
   - `upc_sem_alloc` with an initial non-zero value
   - well-defined error-return semantics for exceeding `UPC_SEM_MAXVALUE`
   - well-defined error-return semantics for destroying a semaphore with concurrent waiters