Proposal for Extending the UPC Libraries with Explicit Point-to-Point Synchronization Support Version 0.2 [*]

Dan Bonachea
Lawrence Berkeley National Laboratory

August 15, 2012

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 pair-wise 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 haven proven useful in a number of UPC applications.

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

[*]This document is a draft - please do not cite in publication.

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 perform more efficiently than an equivalent program expressed directly in UPC.

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 2.2 (8/2005) at 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 are 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).

Implementations that support this interface shall predefine the feature macro `__UPC_SEM__` to the value 1.

2.2 Standard header

The standard header is:

```
    <upc_sem.h>
```

2.3 Semaphore type

The following type is defined in `upc_sem.h`:

```
#include <upc_sem.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-zero 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 *upc_phaseof*(), and *upc_addrfield*() to such pointers are undefined, however *upc_threadof*() may be used to query the thread affinity of a *upc_sem_t* object, which always corresponds to the thread which created the object (via a prior call to *upc_sem_alloc*).

2.4 Semaphore Allocation and Destruction

Synopsis:

```
#include <upc_sem.h>

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

Description:

bupc_sem_alloc non-collectively allocates a semaphore with affinity to the calling thread, initialized to the logical value zero. The *flags* argument indicates the variety of semaphore to be created, which contrains subsequent usage of the object (and possibly allows for greater optimization potential for the more contrained usage models). The flags argument should be a logical OR of the following integral constants, defined in upc_sem.h:

```
UPC_SEM_BOOLEAN
UPC_SEM_INTEGER
UPC_SEM_SPRODUCER
UPC_SEM_MPRODUCER
UPC_SEM_SCONSUMER
UPC_SEM_MCONSUMER
```

UPC_SEM_BOOLEAN / *UPC_SEM_INTEGER* are mutually exclusive, as are *UPC_SEM_SPRODUCER* / *UPC_SEM_MPRODUCER* and *UPC_SEM_SCONSUMER* / *UPC_SEM_MCONSUMER*.

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 increment or decrement a boolean semaphore by more than one. *UPC_SEM_INTEGER* indicates a fully general integer semaphore which may contain any non-negative value not greater than *UPC_SEM_MAXVALUE*, which is guaranteed to be at least 65535.

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. *UPC_SEM_MPRODUCER* allows increments from any thread.

$UPC\_SEM\_SCONSUMER$ 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. $UPC\_SEM\_MCONSUMER$ indicates a semaphore that may be decremented by any thread.

Passing flags == 0 selects most general (safest) configuraton:
$UPC\_SEM\_INTEGER|UPC\_SEM\_MPRODUCER|UPC\_SEM\_MCONSUMER$

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 sempahore are undefined, and if any thread is concurrently accessing the semaphore (e.g., blocked inside *upc_sem_wait*) the behavior is also undefined.

2.5 Point-to-Point Synchronization Operations

Synopsis:

```
#include <upc_sem.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 not less than 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 null strict write operation upon entry to the function, *upc_sem_wait*(N) implies a null strict read operation immediately before exiting, and *upc_sem_try*(N) implies a null strict read operation immediately before a successful completion.

Example 1: The following example demonstrates the framework for a simple producer/consumer program utilizing `upc_sem_t` for synchronization. Even threads produce some data for a partner thread and increment their semaphore, odd threads wait to decrement their semaphore and consume the incoming data.

```
#include <upc_sem.h>

upc_sem_t * shared allflags[THREADS];
shared int applicationData[THREADS];
int tmp;

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

Many applications have a common need whereby a thread must deliver data to a remote thread and simultaneously notify the recipient of arrival. This can be accomplished in UPC 1.2 by combining `upc_memput` with a strict write, but bundling these operations together into a library call can allow more efficient implementations that reduce network traversals.

3.1 Signalling *upc_memput*

Synopsis:

```
#include <upc_sem.h>

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

Description:

Perform a memput with the same data movement semantics as `upc_memput(dst,src,nbytes)`, and increment s by n after the transfer is complete. Requires $upc\_threadof(s) == upc\_threadof(dst)$.

Both functions MAY return on the initiator before the transfer is complete – the semaphore at the target will be atomically incremented by n when the transfer is complete, and the contents of the destination buffer remain undefined until then.

No explicit notifications or guarantees are provided to the initiator regarding the completion of the transfer – the contents of the destination buffer remain undefined until after the semaphore increment. Specifically, upc_fence and other strict operations issued by any thread do not truncate this interval. After a consumer thread successfully returns from decrementing the semaphore, it is guaranteed to observe the updated contents of the destination buffer in subsequent reads.

After a call to *bupc_memput_signal* returns, the source memory is immediately safe to overwrite. *bupc_memput_signal_async* MAY return earlier, while the source memory is still in use by the library (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 initiatior thread, to determine when the source memory is safe to overwrite. Modification of the source buffer prior to such synchronization leads to undefined behavior.

Example 2: *bupc_memput_signal* is demonstrated by replacing the "even thread" branch from Example 1 with the following code:

```
  if (!(MYTHREAD&1)) { /* even threads */

        /* produce some result for my partner thread to a scratch location */
        tmp = computeSomethingInteresting();

        /* send data and notification to my partner */
        upc_memput_signal(&(applicationData[partnerid]), &tmp, sizeof(int),
                          partnerflag, 1);
  } else { // ...
```

4 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. However it's unclear whether a realistic usage scenario can be devised for multiple-consumers, since the successful decrement of a semaphore does not uniquely distribute the protected resource to the consumer (ie thread A and thread B both decremented, now which resource do they consume?)

   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, so we might just disable $UPC\_SEM\_MCONSUMER$ entirely and only provide single-consumer semaphores that must have affinity to the decrementing thread.

2. **Signalling get**
   It might be useful to add a signalling memget analogue, which increments a remote semaphore at the source when the source buffer has been consumed and is safe to overwrite. Probably would be fully blocking at the initiator for dest buffer completion. Would like to see a real motivating example before nailing down semantics.

3. **Allocation Flags**
   Are all the allocation flags useful? Is additional information desireable?

4. **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

5. **Generalized wait/try**
   For a consumer willing/able do from N to M units of work, it might be useful to have a call that atomically performs the following more general wait:

   ```
   int upc_waitNtoM(s, N, M) {
    assert(N<=M);
    while (s->val < N) wait();
    x = MIN(s->val,M);
    s->val -= x;
    return x;
   }
   ```

   and a try variant w/o the wait(). Implementing this, wait and waitN become special cases w/ N=M.