Ideas and Specifications for the new
One-sided Collective Operations in UPC

Version 0.2

Zinnu Ryne, Steven R. Seidel
{zryne, steve}@mtu.edu
Michigan Technological University

December 15, 2005

Traditionally, collective operations have been thought of as two-sided memory operations. This means that while performing such operations, the processing elements (*i.e.,* threads) need to synchronize their execution in order to guarantee certain semantics of the language. Of course, this raises some interesting issues, for example, agreeing on some synchronization point, whether all threads need to synchronize or just the pair of threads, and whether synchronization is needed at all. In the collective operation specification[2, 1] and the draft proposal of the collective extensions[3], this two-sided view has been expressed.

At the 2005 UPC workshop, we have had mixed reactions on the subject of extending the collectives. Here we present a different view of collective operations. To be specific, the proposal suggests looking at collectives as one-sided memory operations, much like `upc_global_alloc` or `upc_free`. Subsetting of UPC threads has been of great concern as that might require major changes in the current language specification. However, the simplified scheme that we present here requires minimal or no changes to the current specification.

The ideas presented in this document are subject to feedback from the UPC community, especially the collectives sub-committee, and are intended to guide our ideas into a complete specification proposal in the near future.

1 Observation

The collective functions are implemented using the `upc_memcpy` family of functions (*i.e.,* either `upc_memget`, `upc_memput`, or `upc_memcpy`). When a call is made, such as `upc_all_broadcast`, what really happens is that after taking care of all synchronization issues, depending on what implementation hint (*i.e.,* "pull" or "push") is provided, threads call `upc_memcpy` to perform the actual data relocation. We observe that `upc_memcpy` does not require the caller to be either the source or the destination of the relocalization operation. This provides an insight into how we can change our view of collectives as one-sided memory operations. We can think of a slightly more complex form of `upc_memcpy` which would provide the compiler or the run-time system (RTS) with just enough information to be able to perform operations analogous to the collective operations.

2 Relocalization Operations

For the purposes of discussion, let us take *broadcast* as an example. When a thread needs to broadcast data the most important things it needs to know is the source and the size of the data. If we could provide

a function that just takes in these two arguments and performs the broadcast operation by figuring out destination addresses on other threads, the function call would look like:

```
upc_broadcast( A, n );
```

where `A` is the source and `n` is its size in bytes. This provides a very elegant design from the user's perspective. However, the specification needs to determine which thread did call the function, where on each thread would the data be copied, and how it would guarantee that data has been copied.

The root of the broadcast is `upc_threadof( &A )`. However, given how `upc_memcpy` works, any thread can initiate the call. Therefore, a single thread calling the function is necessary and sufficient to make the broadcast happen. This solves the first caveat of the specification.

By adding one extra parameter signifying destination, we can eliminate the second caveat, and essentially allow the programmer more control over where they want the data to be distributed on each thread. In that case, the function call would look like:

```
upc_broadcast( B, A, n );
```

where `B` is the destination, `A` is the source and `n` is its size in bytes.

The function simply causes `n` bytes of the block of memory starting at `A` on the root thread to be copied to the corresponding block of memory on each other thread starting at `B`. If the source and destination pointers are same, the broadcast is *in place*.[1]

When the call returns it is safe to write to `A` on the root thread. On all other threads, the copies of `A` will be available at the beginning of the next synchronization phase, however, they will be observable by the calling thread after it returns from the call. If any thread (including the same thread) calls `upc_broadcast` with a source (and destination) that overlap a previous call in the current synchronization phase, then the result is undefined.

Along the same lines we can define both scatter and gather operations. In the following call[2],

```
upc_scatter( B, A, n );
```

`THREADS` blocks of data are scattered to the memory areas corresponding to `B` on each thread. If the source and destination pointers are same, which signals for an `in place` operation, then we need to check for the root thread. If the root is thread 0, then the case is simple. If, however, the root is not thread 0, then the caller's memory area `A` is (carefully) replaced with `A[MYTHREAD]`.[3]

A call to *gather* is simply the inverse of *scatter*. Therefore, any one thread may make the following call:

```
upc_gather( B, A, n );
```

The `gather_all` can be performed by all threads calling `upc_gather`. For example, the functionality can be achieved by the following two calls:

[1] Perhaps, we could simplify this by using a variable number of parameters. For example, if the programmer only specifies the source and size, then we know that the intension is to perform an *in place* broadcast. If the destination is also specified, then we need to check the pointers to see if they are the same; if not, then we have a normal broadcast, otherwise it is *in place*.

[2] It does not matter which thread calls it.

[3] Alternatively, we could move the data to the beginning block of the root thread. However, this raises an issue as to when it is safe to do so. We need to guarantee that the data has already been scattered to the appropriate threads before performing the move.

```
upc_gather( B, A, n );
upc_broadcast( B, B, n*THREADS );
```

Similarly, an exchange can be accomplished by all threads calling `upc_scatter` with suitable arguments. Further, because of the functionality of the permute operation and how it is implemented (with `upc_memcpy`), there is no added advantage of having a specialized function in this case. As far as we can tell, the convenience and potential of optimizations here do not warrant a separate function. However, for convenience and performance, we feel that it is worth providing the `upc_exchange` as a separate function.

3 Computational Operations

Similar to the relocalization operations, we can think of computational operations as being one-sided. For example, let us consider *reduce*.

```
upc_reduce( B, A, TYPE, op );
```

where *TYPE* is the data-type used and *op* is the operation to be performed, which might be a function pointer. We can use this reduce with a one-sided broadcast to build a one-sided `reduce_all`, like the following:

```
upc_reduce( B, A, TYPE, op);
upc_broadcast( B, B, sizeof( TYPE ) );
```

`upc_prefix_reduce` and `upc_xprefix_reduce` (the exclusive version) can be defined in a manner similar to `upc_reduce`.

4 Subsetting

The ideas presented so far suggest an easy way to specify a subsetting feature in one-sided collectives. In simple terms, the threads that do not want to participate in the specific data movement (or computation) operation may simply "sit out" or participate in another operation with other threads as they choose. If we allow the calling thread to specify a list of threads that it wants to participate, it is easy to provide the row-reduce/column-broadcast used in the LU code:

```
upc_reduce(A, type, op, subset[MYROW]);
upc_broadcast(A, sizeof(type), subset[MYROW]);
```

where we have now added a fourth argument `subset[MYROW]` that consists of the thread numbers (perhaps packed into a bit string) of the threads in each caller's row[4]. However, on grounds of ease of usability considerations, we omit the subsetting feature from these sets of functions.

[4]There are two possibilities: we can choose to have a global bit string whenever the program is initialized (both in dynamic and static threads environment). Before every function call which requires a subset of the threads to participate, we can enable/disable bits to signify that the corresponding threads will/will not participate. Or, we can just create an array of threads that will participate in the operation, and pass it to the function.

5 The Proposed API

5.1 One-sided Relocalization Operations

Synopsis

1. `upc_broadcast( shared void * dst, shared const void * src, size_t nbytes )`
   `upc_scatter( shared void * dst, shared const void * src, size_t nbytes )`
   `upc_gather( shared void * dst, shared const void * src, size_t nbytes )`
   `upc_exchange( shared void * dst, shared const void * src, size_t nbytes )`

Description

1. The function treats the `src` pointer as if it pointed to a shared memory area with type:
   `shared [] char [nbytes]` ... for *broadcast*,
   `shared [] char [nbytes*THREADS]` ... for *scatter*,
   `shared [nbytes] char [nbytes*THREADS]` ... for *gather*,
   `shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]` ... for *exchange*

   And it treats the `dst` pointer as if it pointed to a shared memory area with type:
   `shared [nbytes] char [nbytes*THREADS]` ... for *broadcast* and *scatter*,
   `shared [] char [nbytes*THREADS]` ... for *gather*,
   `shared [nbytes*THREADS] char [nbytes*THREADS*THREADS]` ... for *exchange*

2. If `src` and `dst` pointers are same, then the operation is treated as *in place*.

3. The size of elements in each block is `nbytes`, which must be strictly greater than 0; otherwise, no copying will take place.

5.2 One-sided Computational Operations

Synopsis

1. ```
   upc_reduce( shared void * dst, shared const void * src,
       upc_type_t type, upc_op_t op )
   upc_prefix_reduce( shared void * dst, shared const void * src,
      upc_type_t type, upc_op_t op )
   upc_xprefix_reduce( shared void * dst, shared const void * src,
       upc_type_t type, upc_op_t op )
   ```

Description

1. The `dst` and `src` pointers are treated the same way as in the standard reduce and prefix reduce operations.

2. For `type`, see 7.3.2.1.2 of the UPC language specification V1.2.

3. A variable of type `upc_op_t` can have the same values and follows the same restrictions in 7.3.2 of the UPC language specification V1.2.

6 Cost/Benefit Analysis

Here we only reflect on our assumptions.
   ```

- **Cost**:
  We do not foresee any undue burden on the implementors implementing these one sided operations. As we have said, these are only slightly more complex forms of the `upc_memcpy` function. However, synchronization issues need to be carefully sorted out. At this point, we are assuming that the operations guarantee their completion at the end of each synchronization phase.

- **Benefit**:
  There are a number of benefits for the programmer:

  1. The function calls are simple, targeted specifically to "just doing" the operation.
  2. Since these are one-sided, programmers need not worry about explicit synchronization issues.
  3. The operations are by nature asynchronous, as a result we expect better performance compared to the blocking collective operations.
  4. At a first glance, teams seem easy to specify (if not implement) because they are one-sided. The team argument in a non-collective call is just an ordered list of thread numbers.
  5. We anticipate minimal language change (to incorporate the team feature).

7 Concluding Remarks

The ideas presented here are alternatives to the traditional MPI-way of looking at collective operations. Some feedback was generated on this issue from the collectives subcommittee following the UPC workshop at George Washington University. At the Super Computing 2005, the collectives subcommittee generally agreed to have this as a separate specification document.

References

[1] UPC Language Specifications V1.2. [online], June 2005.
`http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf`.

[2] E. Wiebel, D. Greenberg, and S. Seidel. UPC Collective Operations Specification V1.0, December 12 2003.
`http://www.gwu.edu/~upc/docs/UPC_Coll_Spec_V1.0.pdf`.

[3] Z. Ryne and S. Seidel. UPC Extended Collective Operations Specification, August 2005.