S-246, 187

Introduction to UPC and Language Specification

William W. Carlson
Jesse M. Draper
IDA Center for Computing Sciences[1]

David E. Culler
Kathy Yelick
University of California, Berkeley[2]

Eugene Brooks
Karen Warren
Lawrence Livermore National Laboratory[3]

CCS-TR-99-157
May 13, 1999
Second Printing

Abstract

UPC is a parallel extension of the C programming language intended for multiprocessors with a common global address space. A descendant of Split-C [CDG 93], AC [CaDr 95], and PCP [BrWa 95], UPC has two primary objectives: 1) to provide efficient access to the underlying machine, and 2) to establish a common syntax and semantics for explicitly parallel programming in C. The quest for high performance means in particular that UPC tries to minimize the overhead involved in communication among cooperating threads. When the underlying hardware enables a processor to read and write remote memory without intervention by the remote processor (as in the SGI/Cray T3D and T3E), UPC provides the programmer with a direct and easy mapping from the language to low-level machine instructions. At the same time, UPC's parallel features can be mapped onto existing message-passing software or onto physically shared memory to make its programs portable from one parallel architecture to another. As a consequence, vendors who wish to implement an explicitly parallel C could use the syntax and semantics of UPC as a basis for a standard.

1. Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715 (wwc@super.org, jdraper@super.org).
2. Computer Science Division #1776, Soda Hall, University of California-Berkeley, Berkeley, CA 94720-1776 (culler@cs.berkeley.edu, yelick@cs.berkeley.edu)
3. Physics and Space Technology Division, Lawrence Livermore National Laboratory, L-45 LLNL, Livermore, CA 94550 (brooks3@llnl.gov, kwarren@llnl.gov)

1 Introduction

Even though latency and contention problems in large-scale multiprocessors have resulted in a general move away from uniform shared memory toward distributed memory, the shared-memory programming model retains many attractions for users of these systems. In particular, the ability to read and write remote memory with simple assignment statements is considerably more attractive than having to learn all the conventions of a message-passing library, even if the latter is portable. At the same time, the quest for performance often makes it desirable to view program data as distributed among a number of local memories. One of UPC's chief advantages as a language is that it enables programmers to exploit data locality in a variety of memory architectures. UPC does not offer programmers a silver bullet for any parallel programming task. Instead, it assumes that the programmer must think about issues of memory locality in designing effective data structures and algorithms, and it offers the programmer a reasonable means of expressing the result of the design effort.

One of the primary principles of UPC is that the presence of parallelism and remote access should not unduly obscure the resulting program. Users must be able to view the underlying machine model as a collection of threads operating in a common global address space and not worry about such details as whether the model is implemented as shared memory or as a collection of physically distributed memories. Within that model the user makes decisions about data locality and memory consistency. It is up to the compiler and runtime to ensure that the programmer's declarations of shared and private data, and strict or relaxed consistency, are implemented correctly. The programmer's job is to understand the programming model and its relation to the algorithm or application.

To create a memory model in which local data and remote data are differentiated solely by the ways in which they are declared, we have made a small number of modifications to the C language. Because the central question of data locality involves local and remote addresses, we have focused on pointers and arrays, the two C constructs which are most closely tied to addresses. The addition of keywords gives the programmer the ability to distinguish between data that is strictly private to a given thread and data that is shared among all threads in the parallel program. Arrays can be declared to be shared among the threads in a variety of different ways; the result is a flexibility in data layout comparable to that of HPF [HPFF 93].

A UPC program running with shared data on a parallel system will contain at least a single thread per processor. Each thread has local data on which it can operate with all the efficiency of a traditional process on a sequential computer. At the same time, however, it has easy access to shared data that are local to other threads. Unless a user intervenes, there is no implicit synchronization among the threads of computation running in the system.

User intervention in the memory model can happen in several ways. The first involves declarations that tell the compiler how much code motion is acceptable for a given variable; a second involves labels that specify strict or relaxed consistency for a particular statement or sequence of statements. In addition, the user can require synchronization at particular points by specifying barriers, global operations which consist of waiting for the completion of all operations issued before the barrier, followed by an operation which terminates when all the threads have completed these operations.

By keeping software constructs to a minimum and enabling programmers to use underlying hardware efficiently, UPC is solidly in the C tradition. Programmers can write efficient programs because they can choose how much run-

time error checking and synchronization is necessary for their particular codes. At the same time, programmers need to understand the code they are writing. For example, writing a value to a remote memory location does not guarantee that another thread will get the new value when it next reads the variable unless the programmer tells the compiler that such a guarantee is necessary. The compiler is thus free to use the memory consistency mechanisms of the underlying hardware to achieve the best performance it can for the specified program.

2 Shared and Private Data

Adding shared objects to ANSI C required the addition of the keyword "shared" as a *type-qualifier* [ANSI 89] in data declarations to distinguish between declared objects that are shared across all the threads in the system and those that are private to a single thread. In the following declarations, as in all of the example code throughout the paper, `THREADS` is a special `const int` variable which is initialized to the number of threads which the UPC program is using, and `MYTHREAD` is a special `const int` variable which, for each thread, is initialized to the index of that thread, between `0` and `THREADS-1`. Although it is not necessary that shared arrays have a final dimension which is either the symbol `THREADS` or an integer multiple of that symbol, such declarations make explicit the distribution across threads.

```
shared int y[THREADS];          /* one y on each thread */
shared int a[100][THREADS];     /* one a[100] on each thread*/
shared int b[100][12*THREADS];  /* one copy of b[100][12]
                                   on each thread */
shared int x;                /* one x on entire system */
```

2.1 Arrays

As the syntax indicates, the addresses `a[100][5]` and `a[100][6]` are associated with successive threads, with "successive" meaning the next higher numbered thread, with the exception of the last thread, whose successor is thread 0. Note that `b[87][5]` and `b[87][6]` are also on successive threads. The local element of `b` that is one above `b[42][5]` is `b[42][5+THREADS]`. In effect, the layout of `b` is the same as if it had been declared `b[100][12][THREADS]`. The difference between the two is that references to the latter require three subscripts rather than two. A statement of the form

```
i = a[42][i];
```

will result in the computation of an address on thread `i` and a remote fetch from that address. Similarly, a statement of the form

```
a[i][j] = i;
```

will result in the address computation and a remote store to that address.

It is also possible to declare shared arrays that are blocked differently across threads.

```
shared [20] int c[100][THREADS];
shared [11] int d[100][THREADS];
```

Array `c` will be distributed among the threads so that the first 20 elements will be on thread 0, the next 20 on thread 1, and so on. Thus each thread will have the same number of elements of `c` as of `a`, but the arrangement of those elements will be different according to the blocking in the declaration for the arrays. In the case of `d`, the number of elements per thread will not be uniform, but the layout will follow the same basic rule: 11 elements on thread 0, 11 on thread 1, and so on until each of the 100 * THREADS elements has been allocated to a thread.

2.2 Pointers

In order to achieve consistency between arrays and pointers, operations on pointers to shared objects have several characteristics that differ from those of conventional C pointers. Internally, pointers to shared objects have two logically separate components, a thread number and a local address. The thread number is used to determine where the remote reference is to be done, and the local address is used on that thread as if it were in that thread's "local" view. This section discusses several important semantic characteristics of shared pointers, including how they are affected by pointer addition and subtraction and how they are affected by casting to local pointers. It is also important to understand a distinction in the ANSI standard term *type qualifier* [ANSI 89]. The question is "What is shared, the pointer itself, or what it points to?" There are 4 cases:

```
int *p;         /* a local item which points locally */
shared int *p;   /* a local item which points elsewhere */
int *shared p;   /* a shared item which points locally */
shared int *shared p; /* a shared item which points elsewhere */
```

Note that users will find the second example the most useful, but it is possible (as in all C code) to create arbitrarily complex effects by using arbitrarily complex declarators.

2.2.1 Simple Shared Pointer Arithmetic

Pointer arithmetic is defined modulo `THREADS`. For example, successive increments of a shared pointer increment the thread number until it reaches `THREADS` and only then affect the local address. Although shared pointers are not necessarily implemented as structures, structure notation is useful for defining what happens in pointer arithmetic for two shared pointers dp and dp1:

```
shared int *dp, *dp1;
dp1 = dp + i;
```

effectively generates the following code

```
dp1.thread = (dp.thread+i)%THREADS;
dp1.loc = dp.loc+((dp.thread+i)/THREADS)*sizeof(*dp);
```

where `thread` and `loc` refer to the components of this pseudo-structure and `sizeof(*dp)` represents the size of what `dp` and `dp1` point to (int in this case). Thus `dp++` points to the same local address on the next thread unless the

thread number before the increment is `THREADS-1`, in which case `dp++` points to the next local address on thread 0. This convention allows a user to step through an entire shared array one element at a time.

2.2.2 Blocked Pointer Arithmetic

Different conventions govern the use of pointers to blocks of shared data.

```
shared [12] int *p;   /* a local item which points to a block of
                         12 shared ints */
```

These pointers are necessarily more complicated, and their performance may vary considerably from implementation to implementation. Each one contains not only a local address and a thread number, but also some indication of its "phase": where it points in a block of 12. Incrementing the pointer makes it point to successive local addresses on the designated thread until it gets to the end of the block. If the thread number is less than `THREADS-1`, the next increment increases the thread number by one and resets the local address to the beginning of the block and the phase to 0. If the thread number is `THREADS-1`, then the next increment resets the thread number to 0, increments the local address by the size of the base type (int in this case) so that it points to the beginning of the next block, and sets the phase to 0. Because the block specifier is just another form of type qualifier, some rather complex data accesses can be described. For example, consider

```
shared [3] int *shared [5] x[17];
```

This declaration describes an array of 17 shared pointers (blocked in groups of 5), each of which points to shared integers which are blocked in groups of 3. So a reference to x[6][4] would fetch a pointer from the second position of x on thread 1, then add 4 to that pointer (according to the method described above) and fetch the value.

2.2.3 Casting of Shared Pointers to Private Pointers

Whenever a shared pointer is cast to a local pointer, the thread number of the shared pointer is lost. Therefore, this operation is dangerous, because the resulting local pointer may point to a different object than the shared pointer. However, it is useful to get a local pointer for efficient access to the local elements of a shared array.

```
shared int x[THREADS];
int *p;

p = &x[MYTHREAD]; /* p points to x[MYTHREAD], but is
                     more efficient for pointer ops */
```

Casting a private pointer to shared is not allowed in the UPC model. This design decision allows efficient implementation on a wide variety of architectures.

2.3 Consistency Model

UPC provides a hybrid, user-controlled consistency model for the interaction of memory accesses in shared memory space. Each memory reference in the program may be annotated (using a variety of approaches) to be either "strict" or "relaxed". Under "strict" behavior, the program executes in a sequential consistency model [Lamp 79]. This implies that the user can be sure that it appears to *all threads* that the strict references in a thread appear in the order they are written, relative to all other accesses. To implement this, the compiler must take into account all memory accesses in all threads of the program when determining that a strict reference in a thread may be reordered with respect to other shared references. Under "relaxed" behavior, the program executes in what we term a "local" consistency model. This implies that the user can assume that it appears to the *issuing thread* that all shared references in that thread occur in the order they were written. Here the compiler need only analyze shared memory access in the local thread to allow reordering. Note that because each reference may be annotated, a number of models between local and sequential consistency are available to the user.

The general method of using these models is that the programmer will first establish a default environment by including either <upc_strict.h> or <upc_relaxed.h>. Among other things, these files respectively contain a `#pragma upc strict global` and a `#pragma upc relaxed global` directive. All unannotated references within functions defined after these directives operate under the selected model. The programmer then may annotate more explicitly those references to be handled differently. One method for annotating references is to declare shared variables and pointers with the type qualifiers "strict" and "relaxed". All references to annotated variables and through annotated pointers will operate under the selected model, regardless of the default behavior in force. The other method is to use the `#pragma upc strict next` and `#pragma upc relaxed next` mechanisms which apply to the following statement or statement sequence and cause otherwise unqualified references in those statements to operate under the selected mode, without regard to the default behavior in force.

To illustrate these concepts, all of the codes shown in Figure 1 have identical meaning. There are three shared variables, `flag`, `data`, and `data2`, where `flag` is used to protect the shared data in a simple semaphore fashion. We show both the sending and receiving codes (assume that some other mechanism has been used to ensure a single sender and receiver). The effect is the same: the accesses to the two data elements may be rearranged and overlapped to achieve efficiency while the access to `flag` may not. This small example illustrates some of the performance power of the model. If the program were executed entirely under strict semantics, the compiler would most likely be unable to determine that `data1` and `data2` could be reordered (because they are global variables and might be referenced by separately compiled modules, the compiler could not have achieved the performance-enhancing optimization). If the program were executed entirely under relaxed behavior, the compiler could have reordered the flag update, potentially incurring incorrect program results.

2.4 Global Synchronization

UPC provides two mechanisms for global coordination of work in a program: barriers and forall loops. The barrier provided is a split-phase barrier which allows the "notify" phase to be separated from the "wait" phase with local work. This allows for increased barrier efficiency on machines with less than stellar barrier performance. The forall

```
#include <upc_strict.h>
shared int flag;
relaxed shared int data1,data2;

send(int val1,val2) {
   while (flag) /*loop*/;
   data1 = val1;
   data2 = val2;
   flag = 1;
}
int rcv() { int tmp;
   while (!flag) /*loop*/;
   tmp = data1+data2;
   flag = 0;
   return tmp;
}
```

```
#include<upc_relaxed.h>
strict shared int flag;
shared int data1,data2;

send(int val1,val2) {
   while (flag) /*loop*/;
   data1 = val1;
   data2 = val2;
   flag = 1;
}
int rcv() { int tmp;
   while (!flag) /*loop*/;
   tmp = data1+data2;
   flag = 0;
   return tmp;
}
```

```
#include <upc_strict.h>
shared int flag,data1,data2;

send(int val1,val2) {

    while (flag) /*loop*/;
  #pragma upc relaxed next
    {data1 = val1;
     data2 = val2;}
    flag = 1;

}
int rcv() { int tmp;
  while (!flag) /*loop*/
 #pragma upc relaxed next
  tmp = data1+data2;

  flag = 0;
  return tmp;
}
```

```
#include<upc_relaxed.h>
shared int flag,data1,data2;

send(int val1,val2) {
#pragma upc strict next
    while (flag) /*loop*/;
    data1 = val1;
    data2 = val2;
#pragma upc strict next
    flag = 1;

}
int rcv() { int tmp;
#pragma upc strict next
  while (!flag)/*loop*/;
  tmp = data1+data2;
#pragma upc strict next
  flag = 0;
  return tmp;
}
```

Figure 1. Examples of Memory Consistency Models in UPC

construct allows not only the specification of work sharing, but also the affinity of data and work assignment for increased efficiency.

2.4.1 Barriers

A barrier synchronization point in a parallel program has the effect of causing all threads to wait at the barrier until every thread has reached it. To accomplish this behavior, there are two distinct actions each thread must take: notifying all other threads that it has reached the barrier and waiting for every other thread to report its presence. Whether barriers are implemented in hardware or software on a given machine, these steps are always required and often distinct. UPC provides three routines to allow users to implement barriers: `upc_notify()`, `upc_wait()`, and `upc_barrier()`, the final being a convenience notation which combines notify and wait. The advantage of this design is that it allows users (or compilers) to place local work between the notify and wait calls. UPC compilers (or programmers) can be free to move local work with no potentially shared writes which occur before a notify to after it and local work with no potentially shared reads occurring after a wait to before it. In the equivalent codes shown in Figure 2, the code on the right is potentially more efficient because it allows the local work. The performance of the

```
int x;                                  int x;
shared int *p;                          shared int *p;

x = *p;                                 x  = *p;
x += local_pure_function();             _upc_notify();
_upc_barrier();                          x += local_pure_function();
*p = x;                                 _upc_wait();
                                        *p = x;
```

Figure 2. Split-phase Barriers Allow Local Computation

split code will be greater than that of the barrier code because the variance in arrival time at the notify point will be covered by the time to execute the local function. If the variance is less than the time taken for the local function, no processor will need to wait at the wait point.

2.4.2 Forall

The forall statement allows programmers to describe a global loop with assignment of threads to loop indices made by the compiler and/or runtime system. It makes a fair number of assumptions about the environment of the system when the loop is executed, including that all threads enter the outermost forall statement of a set of nested forall statements. Users writing “SPMD” style programs should find this convenient. Because forall is also a parallel statement, all iterations of the loop are independent and can be executed in any order desired by the compiler and runtime system. The forall statement adds to the traditional for statement a fourth field that describes the affinity under which to execute the loop. In a typical case, the affinity expression will be an expression which occurs in the loop. Consider the following code:

```
shared float x[100], y[100], z[100];

addit()
{
    int i;
     upc_forall (i=0; i<100; i++; &x[i]) {
       x[i] = y[i] + z[i];
    }
}
```

In this code, the compiler can assign tasks to threads so that `x[i]` is local to that thread. It can also detect from the data declaration that `x`, `y`, and `z` are aligned, so it can arrange to execute this code with total locality. The forall statement has no implied barriers; if barriers are needed before or after the forall loop to ensure proper program execution, they need to be inserted.

3 Implementation

UPC has been designed so that it can be implemented efficiently on a variety of multiprocessor architectures, including shared memory systems as well as distributed shared memory systems. While the details of implementation may vary considerably from system to system, the primary focus for the user should be on the semantics defined in previous sections. For example, a shared array `x` will be laid out in memory so that thread 0 will be able to access `x[0]` and `x[THREADS]` equally well. On a distributed shared memory system such as the T3E these two elements will be adjacent in the local memory of the processor that executes thread 0. On a shared memory machine the thread number may become part of a virtual address in such a way that `x[0]` and `x[THREADS]` will be adjacent on a page of physical memory in some way assigned to thread 0. On any system there are two primary issues to discuss: how user variables are laid out in system memory, and how pointers are dereferenced so that address arithmetic functions correctly.

3.1 Thread and Memory Layout

In distributed memory systems, such as the T3D and T3E, threads are assigned to processors in a one-to-one mapping. Each processor has an inherently separate address space, so private variables are laid out as they would be on a uniprocessor system. Shared scalars are all allocated to thread zero's memory, while shared arrays (and memory dynamically allocated through `upc_alloc()`) are spread across the system, so that each processor has a `1/THREADS` portion of the array, subject to the blocking specified in the declaration. In the case of blocking size of 1 (the default), consecutive elements are assigned to consecutive threads.

In shared memory systems, threads are also mapped to distinct processors; however, there may be reasons to map more than one thread to each processor. Operating systems for shared memory systems support a variety of shared and private data spaces among threads, requiring somewhat different data mappings. The most advantageous model (from UPC's perspective) is one in which each thread of a process can map (using the virtual memory system) its entire space twice: once for private accesses (which is usually achieved by copy-on-write) and once for shared access among all threads. If such mappings are unavailable, another model (e.g., MPI) causes all data to be private. On shared-memory implementations of MPI, programs may allocate (e.g., mmap) shared memory after thread invocation. Finally, all shared memory machines support a model (e.g., pthreads) which causes all static data to be shared and all automatic data to be private. The mmap style of implementation was used successfully on several platforms in PCP [BrWa95]. Many systems allow any of the three models to be used.

On the model which allows both private and shared views of the same memory, UPC requires no special runtime allocation or transformations. As in the distributed memory implementation, shared accesses are made through the shared virtual memory mapping; private accesses, through the copy-on-write virtual addresses. To achieve high performance under the UPC memory model, the addressing of shared objects needs to be transformed so that all references to shared data which have consecutive addresses in the "thread-affinity" space are accessed at consecutive addresses in the shared virtual memory system. This will usually involve mapping the "thread" field of a distributed pointer into the high order bits in a virtual address.

Under the MPI-like model, UPC will allocate private data without special transformations. Shared static data will be allocated at program startup to a shared area and references to this data will be appropriately transformed by the UPC

compiler. Finally, under the pthreads-like model, UPC will allocate shared static data and private automatic data without special attention. To accommodate private static data, the UPC compiler will transform the static references into equivalent references to private data allocated at startup.

Table 1 summarizes allocation and access types for various systems. In most cases, the table specifies that no special attention is needed for local data accesses, which will lead to highly efficient code. Note that hybrid shared/distributed memory systems (e.g., "clustered SMP's") can easily be handled by running multiple threads per shared memory node under a shared-memory model, and then combining them using distributed memory techniques. In such a case UPC would provide a unified shared-memory programming model across the complex system.

Type	**DSM**	**Shared Mem**		
		mapping	**MPI-like**	**pthreads-like**
private-static	uniprocessor	uniprocessor	uniprocessor	allocate/transform
private-auto	uniprocessor	uniprocessor	uniprocessor	uniprocessor
private-heap	malloc	malloc	malloc	malloc
shared-static	`1/THREADS`:get/put	shared segment	allocate/transform	uniprocessor
shared-heap	upc_alloc:get/put	upc_alloc	upc_alloc	upc_alloc

TABLE 1. Data Mapping and Access

3.2 An Example Implementation of Pointers to Blocked Shared Memory

Each implementation of blocked shared pointers will have to balance efficiency of remote accesses with efficiency of pointer arithmetic. The initial implementation on the T3E makes a small sacrifice in the efficiency of pointer arithmetic in order to keep gets and puts as efficient as those using normal shared pointers. Figure 3 shows the details of

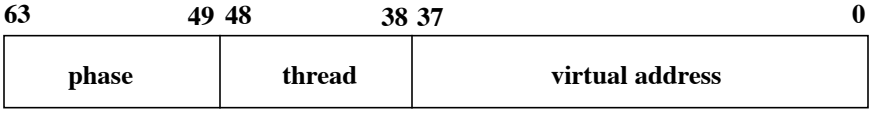

Figure 3. Blocked Pointer Implementation

this implementation. The bottom 49 bits of this pointer are identical to those of an unblocked shared pointer; as a consequence, the compiler can simply zero the top 15 bits and use the resulting pointer for a remote access. Pointer arithmetic requires more work. If we think of the pointer as a structure containing fields "phase", "thread", and "virtual address", then the following code describes what the compiler computes for pointer arithmetic.

```
shared int *sp, *sp1;
sp1 = sp + i;
```

effectively generates

```
sp1.phase = (sp.phase+i)% BLOCK;
sp1.thread = (sp.thread + ((sp.phase+i)/ BLOCK))% THREADS;
```

```
sp1.vadr = sp.vadr + (sizeof(*sp) * (sp1.phase - sp.phase + (BLOCK
* ((sp.thread + ((sp.phase+i)/ BLOCK))/ THREADS))));
```

In this case BLOCK is the integral number of elements specified as the block size in the declaration of the variable. On the T3E thread is just the processor number, and `THREADS` is the number of processors in the job. Of course, all of the pointer arithmetic will be more efficient when both the block size and the number of processors are powers of 2. As the bit layout in Figure 3 shows, this particular implementation also imposes a limit of 2^{15} on block sizes.

4 References

[ANSI 89]American National Standards Institute, *American National Standard for Information Systems--Programming Language--C*, 1989, sec. 3.5.4.1.

[BrWa 95] Brooks, Eugene, and Karen Warren, "Development and Evaluation of an Efficient Parallel Programming Methodology, Spanning Uniprocessor, Symmetric Shared-memory Multi-processor, and Distributed-memory massively Parallel Architectures," poster session at *Supercomputing '95*, San Diego, CA, December 3-8, 1995.

[CaDr 95]Carlson, William W. and Jesse M. Draper, "Distributed Data Access in AC," *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Santa Barbara, CA, July 19-21, 1995, pp. 39-47.

[CDG 93]Culler, David E., Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick, "Parallel Programming in Split-C," in *Proceedings of Supercomputing '93*, Portland, OR, November 15-19, 1993, pp. 262-273.

[HPFF 93]High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.0*, Rice University, May 1993.

[Lamp 79]Lamport, L., "How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs," *IEEE Transactions on Computers,* Vol. C-28, No 9, September 1979.

[Stal 94]Richard M. Stallman, *Using and Porting GNU CC*.

5 Appendix: UPC Language Specification, Version 0.91

UPC is an extension to ANSI C [ANSI 89] which supports a simple model of shared memory parallel programming and allows simultaneous ease of programming and high performance. This appendix is intended to describe the syntactic and semantic extensions to the ANSI C document necessary to achieve this result. It assumes a thorough knowledge of the ANSI C specification and the language concepts discussed elsewhere in this paper. Just as with ANSI C it has both forward and backwards references, so several reads are probably appropriate.

5.1 Program Execution

A UPC program, like an ANSI C program, operates in a well-defined abstract execution model. This model is one in which THREADS units of independent computation are started, each calling the UPC program's main() routine. Unless declared objects or references are qualified as both shared and strict, there is no change to the ANSI C execution model applied to an individual unit of computation. This implies that translators are free to reorder and/or ignore operations as long as the restrictions of the model described in section 2.1.2.3 of the ANSI C specification are observed.

We add a further restriction on this model to handle strict shared references. Define the order of shared references in each thread of computation as that specified by the programmer. For each strict shared reference, the output to files of all threads of computation must not change if that reference is eliminated or reordered (by any compiler or runtime implementation) with respect to all other shared references in the thread.

There is an implied strict shared reference before a `upc_notify` statement and after a `upc_wait` statement, as well as before each thread begins and after it ends.

5.2 DECLARATIONS

UPC extends the declaration ability of C to allow for shared types, enhanced data layout, and ordering constraint specification.

5.2.1 Syntax

```
type_qualifier_1: shared layout_specifier
                  strict
                  relaxed

layout_specifier: null
                  layout_specifier [ expression ]
```

5.2.2 Constraints

The same qualifier may not appear more than once in the same specifier list or qualifier list, either directly or via one or more typedefs.

The same specifier list or qualifier list may not contain both `strict` and `relaxed`.

`strict` or `relaxed` may not appear if `shared` does not.

`shared` may not appear in the specifier-qualifier-list of a structure-declaration or an object which has automatic storage duration, unless it qualifies an item pointed at.

`shared` may not appear in a type cast whose object is not shared qualified.

5.2.3 Semantics

Shared-qualified types refer to objects in a common address space shared by all threads. This implies that, for example, any thread may reference statically declared shared objects and these references refer to the same object. Also, any thread may take the address of a shared object, pass it in some way to another thread, and when that thread dereferences the pointer, it will be the same object referred to by the original thread. Types which are not shared-qualified are in a private address space, and references to their addresses by another thread have undefined behavior.

If a reference to a shared-qualified object is cast to a non-shared-qualified reference, and the underlying object is dereferenced by a thread without affinity (defined below) to the object, the result is undefined.

A shared reference which is cast to non-shared will be interpreted as implicitly non-shared and need only obey the semantics of thread ordering, independent of the strict-qualified references referred to above.

References which are neither strict nor relaxed qualified behave as strict or relaxed based on the default translation behavior, or if none is specified, on an implementation-defined basis.

5.3 Data Layout

Shared-qualified objects are placed in memory based on an affinity to a particular thread. This affinity is defined based on the ability to refer to the object by a cast to a non-shared-qualified reference.

All non-array shared-qualified objects have affinity with thread zero.

The affinity of element I of an array object is defined in terms of the “block size” and “number of threads”. The block size is the product of the elements of the layout_specifier. The number of threads is a function of the runtime environment. Element I of an array object has affinity with thread (floor(I/block_size)%THREADS). The default layout specifier is [1].

5.3.1 Examples

```
shared int x;    /* a single integer, with affinity to thread 0 */
shared float y[4];  /* an array of floats, with affinity to
                       threads 0,1,2, and 3 respectively, if
                       THREADS >= 4 */
shared [3] struct a z[5]; /* an array of structures: z[0],z[1],z[2]
                             have affinity to thread 0; z[3],z[4]
                            have affinity to thread 1 */
shared int *p;      /* a non-shared pointer which points at
                       a shared object */
shared int *shared p; /* a shared pointer which points at
                         a shared object */
```

One can get arbitrarily complex:

```
shared [3] int *shared[5] x[7];
```

x is an array of shared pointers (blocked in groups of 5), each of which points to shared integers, blocked in groups of 3

5.4 The NULL strict reference

Note that the equivalent of a "fence" or "remote memory barrier" may be achieved by a "null" strict reference:

```
{static shared strict int x; x=x;}
```

This reference's execution can be eliminated (it has no scope and can therefore have no effect on program output), but the fact that the shared reference exists in the abstract sequence cannot be removed (as it may have effect on program output). The construct will therefore act solely as a fence for the shared references occurring before or after it. This construct is provided as the `upc_fence()` macro.

5.5 Default Behavior Control

5.5.1 Syntax

```
#pragma upc strict global
#pragma upc strict next
#pragma upc relaxed global
#pragma upc relaxed next
```

5.5.2 Semantics

These directives control the default behavior of functions that follow. When operating under a strict default, the effect of all accesses to non-qualified shared objects are considered to be strict qualified; under relaxed default, accesses to non-qualified shared objects are considered to be relaxed qualified.

The next qualifier means the default applies only to the next C statement. Any global default is restored after that statement. The global qualifier means the specified behavior (strict or relaxed) applies to the remainder of the translation unit.

Shared objects which are explicitly qualified as either strict or relaxed are unaffected by these pragmas.

5.6 Barrier Statements

5.6.1 Syntax

```
stmt_1:     upc_notify barrier_value
            upc_wait barrier_value
            upc_barrier barrier_value

barrier_value:   null
                 expr
```

5.6.2 Constraints

Each thread of computation must execute an alternating sequence of `upc_notify` and `upc_wait` statements, starting with a `upc_notify` and ending with a `upc_wait` statement.

5.6.3 Semantics

Barrier is a thread synchronization mechanism. Its semantics are defined operationally. A thread will not proceed past the `upc_wait` statement until all other threads in the program have entered a `upc_notify` statement.

If any thread's `barrier_value` is different from either its own (e.g., `upc_notify 3; upc_wait 7`) or that of any other threads, a runtime signal is produced (SIGURG). The default value of `barrier_value` is the integer zero. This value is solely for use in debugging, to insure that all threads reach the same barrier, if desired. A zero value matches all others.

If a thread executes a `upc_barrier`, `upc_notify`, or `upc_wait` within the dynamic scope of a forall statement, the result is undefined.

The `upc_barrier` statement is equivalent to the compound statement:

```
{upc_notify barrier_value; upc_wait barrier_value;}
```

5.7 Forall Statement

5.7.1 Syntax

```
stmt_2:  upc_forall ( expr1 ; expr2 ; expr3 ; affinity ) statement

affinity:    null
             expr
             continue
```

5.7.2 Semantics

This statement is equivalent to the following:

```
for (expr1 ; expr2; expr3)
   if (upc_threadof(affinity) == MYTHREAD)
     statement;
```

Unless all threads enter the outermost `upc_forall` nest, the behavior is undefined. The threads may enter or leave in any order.

If the body statement contains side-effects visible to expr1, expr2, or expr3, the behavior is undefined.

If the body statement contains any inter-loop data-dependencies, the behavior is undefined.

If the body statement contains any variant of `upc_barrier`, the result is undefined.

If the body (or entry) has side-effects on non-shared objects, the value of these objects after the loop is undefined

If the affinity expression is “continue”, the body statement must contain a `upc_forall` statement, with defined affinity.

5.8 Library/Builtin functions

```
shared void *upc_local_alloc(int size, int nmemb);
shared void *upc_global_alloc(int size, int nmemb);
shared void *upc_all_alloc(int size, int nmemb);
void upc_free(shared void * ptr);
```

These routines allocate memory which is shared. Each returns a shared pointer to at least `size * nmemb` bytes which is suitable for assignment to a shared pointer of a compatible size. `upc_local_alloc` returns a pointer to memory in which every element has affinity with the caller’s thread. `upc_global_alloc` and `upc_all_alloc` return pointers to memory which is equivalent to that which is allocated for a static shared array of `nmemb` elements of the same size. `upc_global_alloc` is called by a single thread, while `upc_all_alloc` must be called by all threads and allocates a single shared array of size `size * nmemb` bytes. `upc_free` deallocates memory.

```
int upc_threadof(shared void *ptr);
```

Returns the thread which `*ptr` has affinity.

```
void upc_lock(shared upc_lock_t l);
int upc_lock_attempt(shared upc_lock_t l);
void upc_unlock(shared upc_lock_t l);
```

These routines implement simple shared locks. `upc_lock` spins until the lock succeeds. `upc_lock_attempt` makes a single attempt to lock and returns 1 on success, 0 on failure. `upc_unlock` releases the lock.

5.9 upc_strict.h and upc_relaxed.h

upc_strict.h is:

```
#pragma upc strict global
#include <upc.h>
```

upc_relaxed.h is:

```
#pragma upc relaxed global
#include <upc.h>
```

upc.h is:

```
#define barrier upc_barrier
#define barrier_notify upc_notify
#define barrier_wait upc_wait
#define forall upc_forall
#define fence upc_fence
```