A Brief Introduction to UPC

William W. Carlson[a]

[a]IDA Center for Computing Sciences, 17100 Science Drive, Bowie MD 20715
wwc@super.org

UPC is a parallel extension of the C programming language which provides programmers with a shared global address space. A descendant of Split-C [1], AC [2], and PCP [3], 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 Cray 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, UPC has seen implementations on a variety of hardware platforms and is becoming a standard language.

UPC est une extension parallèle du langage C, qui donne aux programmateurs un accès mémoire global. UPC est un héritier de Split-C [1], AC [2], et PCP [3], et vise deux objectifs : 1) donner un accès efficace à la machine sous-jacente; 2) établir une syntaxe et une sémantique communes pour la programmation explicitement parallèle en C. La recherche d'une performance maximale fait que UPC tente de minimiser la surcharge résultant de la communication entre les processus. Lorsque le matériel permet à un processeur d'accéder à de la mémoire distante sans l'intervention d'un processeur distant (comme chez le Cray T3E), UPC donne au programmateur une correspondance simple et directe entre le langage et les instructions-machine de bas-niveau. D'autre part, les caractéristiques parallèles de UPC correspondent à des librairies de passage de message existantes ou à des appel à une mémoire partagée, ce qui rend les programmes portables d'une architecture parallèle à une autre. UPC est implémenté sur plusieurs plate-formes et est en train de devenir un langage standard.

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 [4].

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 runtime 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 Threads

In UPC, a program consists of some number of threads of computation, each of which behaves like a normal C program. They all call the program's `main()` and each ends by calling `exit()` or by returning from `main()`. In UPC programs, `THREADS` is an integer value representing the number of threads which the UPC program is using, and `MYTHREAD` is an integer value which, for each thread, is the index of that thread, between 0 and `THREADS`-1. These values are automatically available to all UPC programs.

3 Shared and Private Data

Adding shared objects to ANSI/ISO C required the addition of the keyword `shared` as a type-qualifier [5] 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. Examples of shared and private declarations include:

```
/* one xp private to each thread */
int xp;

/* one xs shared by all threads */
shared int xs;

/* one int per thread, shared by all */
shared int y[THREADS];

/* one a[100] per thread*/
shared int a[100][THREADS];

/* no need to be a multiple of THREADS */
/* a total of 100 ints, shared by all */
shared int b[100];
```

3.1 Affinity

Each element of a shared object has *affinity* to a single thread. While any thread may access any element of a shared object, the thread with affinity may treat the element as private, often getting a performance boost. In the simple case of arrays ending in a `THREADS` dimension, the syntax indicates affinity; 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.

A statement of the form

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

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

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

will result in the address computation and a store to address i on thread j.

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

```
shared [20] int c[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.

3.2 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. It is important to understand a distinction in the ANSI standard term type qualifier[5]. The question is "What is shared, the pointer itself, or what it points to?" There are 4 cases:

```
/* a private pointer to a private object */
/* THREADS copies, all independent */
int *p2p;

/* a private item which points to shared */
/* THREADS copies, all independent */
shared int *p2s;

/* a shared item which points to a private */
/* single object, shared by all */
int *shared sp2p;

/* a shared item which points to a shared */
/* single object, shared by all */
shared int *shared sp2s;
```

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.

3.3 Casting

Whenever a pointer to shared is cast to a pointer to private, the thread number is lost. Therefore, this operation is dangerous, because the resulting pointer may point to a different object than the input pointer. However, it is useful to get a pointer to private for efficient access to the local elements of a shared array. Such casting is only allowed when the pointed-to shared object has affinity with the casting thread.

```
\begin{verbatim}
shared int x[THREADS];
int *p;
```

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

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.

3.4 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 similar manner to a sequential consistency model [6]. This implies that the user can be sure that it appears to all threads that the strict references in a particular thread appear in the order they are written, relative to all other accesses. 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. Note that because each reference may be annotated, a number of models between relaxed and strict 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>`. 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.

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

3.5.1 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 statements which 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.

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

x = *p;           x  = *p; upc_notify;
upc_barrier;      x += local_pure_function();
*p = x;           upc_wait; *p = x;
```

The performance of the 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.

3.5.2 Forall

The `upc_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 `upc_forall` statement of a set of nested `upc_forall` statements. Users writing "SPMD" style programs should find this convenient. Because `upc_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 `upc_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 `upc_forall` statement has no implied barriers; if barriers are needed before or after the loop to ensure proper program execution, they need to be inserted.

4 Implementation

UPC has been 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.

Acknowledgments

This paper represents the efforts of a large number of people working on the UPC team over the past 5 years. The language was designed by Eugene Brooks, William Carlson, David Culler, Jesse Draper, Karen Warren, and Katherine Yelick. Industry teams from Cray, Compaq, Computer Sciences Corp, Etnus, Intrepid, Hewlett-Packard, IBM, and SGI, and Sun have all worked on implementations of UPC. Research teams from the Arctic Region Supercomputer Center, George Washington University, The IDA Center for Computing Sciences, Lawrence Berkeley National Labs, Lawrence Livermore National Labs, Michigan Technological University, and The University of California, Berkeley have all contributed greatly to UPC’s ongoing development. See the website http://www.upcworld.org for more details. The website also includes information on available libraries, implementations, documentation, and tutorials.

References

1. David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eiken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing ’93*, pages 262 – 273, Portland, OR, November 1993.
2. William W. Carlson and Jesse M. Draper. Distributed data access in AC. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 39 – 47, Santa Barbara, CA, July 1995.
3. Eugene D. Brooks and Karen Warren. Development of an efficient parallel programming methodology, spanning uniprocessor, symmetric shared-memory multiprocessor, and distributed-memory massively parallel architectures. *Supercomputing ’95*, 1995.
4. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*. Rice University, May 1993.
5. ANSI. *Programming Languages - C*. ISO/SEC9899, 2000.
6. L Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9), September 1979.