Draft

v 0.1r

THE GEORGE WASHINGTON UNIVERSITY

High Performance Computing Laboratory

UPC Manual

THE GEORGE WASHINGTON UNIVERSITY

UPC Manual

Sébastien Chauvin
Proshanta Saha
François Cantonnet
Smita Annareddy
Tarek El-Ghazawi

The George Washington University
801 22nd Street NW • Suite 607
Washington, DC 20052

Table of Contents

1 Introduction to UPC 5
1.1 What is UPC? 5
1.2 UPC Programming Model 6
1.3 The UPC Memory model 6
1.4 Data Distribution and Coherency 7
1.5 Collective Operations 8
1.6 Parallel I/O 8
2 Programming in UPC 9
2.1 My First UPC Program 9
2.2 Most Commonly Used Keywords and Functions 10
2.2.1 THREADS 10
2.2.2 MYTHREAD 10
2.2.3 upc_forall 11
2.2.4 shared 12
2.2.5 upc_barrier 14
2.2.6 upc_lock/upc_unlock 15
2.3 Memory Consistency 17
2.4 Compiling and Running UPC Programs 18
2.5 Vector Addition Example 19
3 Data and Pointers in UPC 22
3.1 Shared and Private Data 22
3.2 Blocking of Shared Arrays 23
3.3 UPC Shared and Private Pointers 25
3.4 Shared and Private Pointer address format 26
3.5 Special UPC Pointer Functions and Operators 26
3.6 Casting of Shared to Private Pointers 27
3.7 UPC Pointer Arithmetic 28
3.8 UPC String Handling Functions 29
4 Work Distribution 32
4.1 Data Distribution for Work Sharing 32
5 Synchronization and Memory Consistency 34
5.1 Barrier Synchronization 34
5.2 Synchronization Locks 37
5.3 Ensuring Data Consistency 39
6 Dynamic Memory Allocation in UPC 43
6.1 Dynamic Shared Memory Allocation 43
6.2 Freeing Memory 45
6.3 Memory Allocation Examples 46
7 UPC Optimization 47
7.1 How to Exploit the Opportunities for Performance Enhancement 47
7.2 Compiler and Runtime Optimizations 47
7.3 List of Hand Tunings for UPC Code Optimization 48
7.3.1 Using local pointers instead of shared pointers 48
7.3.2 Aggregation of Accesses Using Block Copy 48
7.3.3 Overlapping Remote Accesses with Local Processing 49

8 UPC Programming Examples 51
8.1 Sobel Edge Detection 51
8.2 N-Queens 54
Appendix A: Programmer's Reference Guide 58
Reserved Words in UPC 58
Libraries and Headers 59
UPC Keywords 59
Shared variable declaration 59
Work sharing 60
Synchronization 61
UPC operators 62
Dynamic memory allocation 62
String functions in UPC 62
Locks 62
General utilities 63
Appendix B: Running UPC on implementations 64
Available Compilers 64
Compiling and Running on Cray T3E 64
Compiling and Running on HP/Compaq 64
Compiling and Running on SGI 65
Compiling and Running on Berkeley UPC 65
Appendix C: Performance Tuning 66
Runtime optimizations on HP (1) 66
Runtime optimizations on HP (2) 66
How to set an environment variable 67
SMP local optimization 67

About this manual

This UPC manual is intended for all levels of users, from novices who wish to find out features of UPC that may fit their needs, all the way to experts who need a handy reference for the UPC language. It is important to note that this is not a replacement for the UPC Specifications document[ElG03a], upon which the UPC language is built. This manual will introduce the basic concepts most commonly asked about by UPC users. For more intricate details of the language, it is best to refer to the UPC Specifications document. This manual is written under the assumption that the reader understands and is familiar with the C language, and has at least some basic knowledge of parallel programming models such as Message Passing or Shared Memory.

Web page

The UPC distributions, tutorials, specifications, FAQ and further documentation are available at the UPC web page:

http://upc.gwu.edu

Support

Please send questions and comments to

upc@gwu.edu

Mailing lists

To join the UPC-users mailing list, visit the UPC web site at:

http://upc.gwu.edu

Chapter

1

1 Introduction to UPC

This chapter provides a quick overview of Unified Parallel C (UPC). Discussions include a brief introduction to the roots of UPC, the UPC memory and programming models, compilers and run time environments. The language features of UPC will be examined in chapters 2 to 6.

1.1 What is UPC?

Unified Parallel C (UPC) is an explicit parallel extension of ANSI C and is based on the partitioned global address space programming model, also known as the distributed shared memory programming model. UPC keeps the powerful concepts and features of C and adds parallelism; global memory access with an understanding of what is remote and what is local; and the ability to read and write remote memory with simple statements. UPC builds on the experience gained from its distributed shared memory C compiler predecessors such as Split-C[Cul93], AC[Car99], and PCP[Bro95]. The simplicity, usability and performance of UPC have attracted interest from high performance computing users and vendors. This interest resulted in vendors developing and commercializing UPC compilers. UPC is the effort of a consortium of government, industry and academia. Through the work and interactions of this community, the UPC Specification V1.0 was produced in February 2001. Subsequently, UPC Specification V1.1 was released in May 2003. Compilers for HP, SGI, Sun and Cray platforms have been released. Open-source implementations

are also available from Michigan Technological University (MuPC) for the 32bit Intel Architecture, and from University of California Berkeley (BUPC) for various platforms including the 32bit and 64bit Intel Architectures. An open-source implementation for the Cray T3E also exists. There are many other implementations underway.

1.2 UPC Programming Model

UPC utilizes a distributed shared memory programming model. The distributed shared memory model is similar to the shared memory model with the addition of being able to make use of data locality. The distributed shared memory model divides its shared address space into partitions where each memory partition M_i has affinity to thread Th_i, see figure 1.2-1. Here affinity indicates in which thread's local shared memory space a shared object will reside.

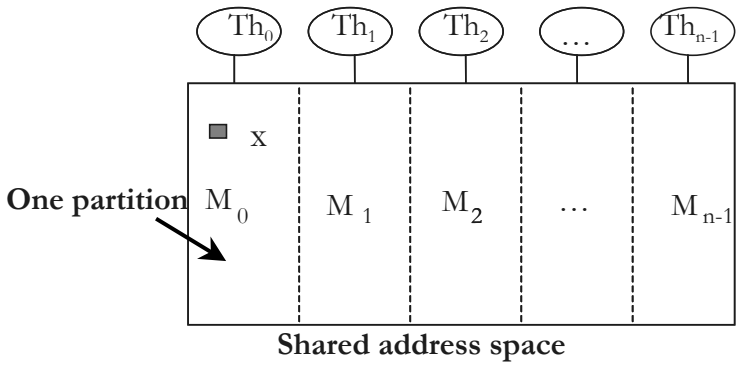


Figure 1.2-1 Shared Memory Model

Some of the key features of UPC include the use of simple statements for remote memory access, efficient mapping from language to machine architecture, and minimization of thread communication overhead by exploiting data locality. UPC introduces new keywords for shared data and allows easy blocking of shared data and arrays across the executing threads.

1.3 The UPC Memory model

The UPC memory view is divided into private and shared spaces. Each thread has its own private space, in addition to a portion of the shared space. Shared space is partitioned into a number of

partitions each of which has affinity with a thread, in other words it resides on the thread's logical memory space as seen in figure 1.3-1.

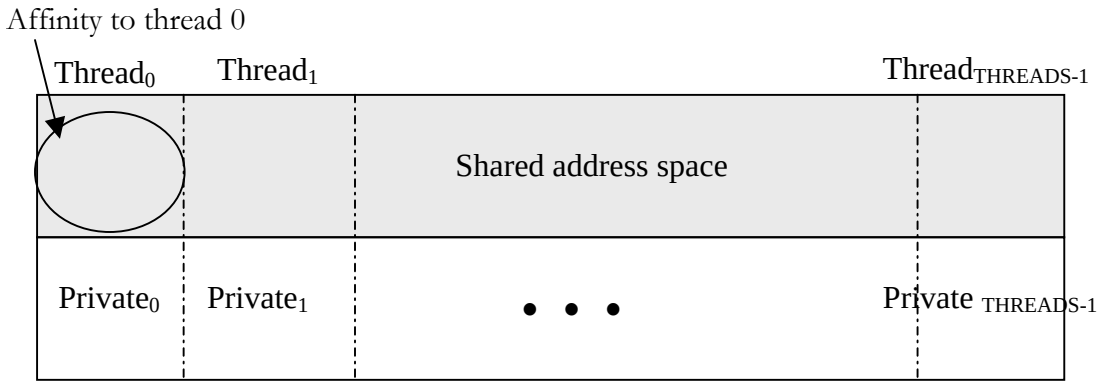


Figure 1.3-1 Thread Affinity

A UPC shared pointer can reference all locations in the shared space; while a private pointer may reference only addresses in its private space or in its local portion of the shared space. Static and dynamic memory allocations are supported for both shared and private memory.

1.4 Data Distribution and Coherency

Data distribution in UPC is simple due to the use of the distributed shared memory programming model. This allows UPC to share data among the threads using simple declaration statements. To share an array of size N equally among the threads the user simply defines the array as a `shared`, and UPC will distribute the array elements in a round robin fashion. Details and semantics of data distribution can be found in chapter 3.

Since shared memory is accessible by all the threads, it is important to take into consideration the sequence in which memory is accessed. To manage the access behaviors of the threads, UPC provides several synchronization options to the user. First the user may specify strict or relaxed memory consistency mode at the scope of the entire code, a section of a code, or to an individual shared variable. Secondly the user may use locks to prevent simultaneous access by more than one thread. Thirdly, the user may use barriers to ensure that all threads are synchronized before further

action is taken. More detailed discussions and semantics on data coherency and synchronization can be found chapter in Section 2.3 and in Chapter 5.

1.5 Collective Operations

Work on the collective operation specifications is currently under progress to define library routines which provide performance enhancing relocalization and data parallel computation operations. The end result is to have functionality often referred to as "collective operations" provided in a manner appropriate to UPC's practical performance model. For more details please refer to the UPC Collective Operations working group's latest specification document version 1.0 pre-release revision 4[Eli03].

1.6 Parallel I/O

Effort is currently under way to build an application-programming interface (API) for parallel I/O in UPC, known as UPC-IO. The UPC-IO API is designed in a manner consistent with the spirit of the UPC language to provide application developers with a consistent and logical programming environment. This effort leverages the wealth of work that has already been performed by the parallel I/O community, particularly the parallel I/O API in MPI-2, commonly known as MPI-IO[MPI2]. For more details please refer to the UPC Parallel I/O working group's latest specification document version 1.0 pre-release revision 9[ElG03b].

Chapter

2

2 Programming in UPC

This chapter focuses primarily on introducing the subtle differences between UPC and C, encompassing syntax, keywords, and necessary headers. The following few chapters provide the concepts of parallel programming in UPC.

2.1 My First UPC Program

Example 2.1-1 shows a simple UPC program whose output shows the manner in which threads in UPC work. The program is executed by all threads simultaneously. The program prints a "`Hello World`" statement in addition to stating which thread is running the program and how many threads are active in this program.

Example 2.1-1:

```
1:     #include <upc_relaxed.h>
2:     #include <stdio.h>
3:
4:     void main(){
5:          printf("Hello World from THREAD %d (of %d THREADS)\n",
6:          MYTHREAD, THREADS);
7:     }
```

Line 1 of the program specifies the memory consistency mode which the program runs under. In this example the program runs using the relaxed mode as specified by the `upc_relaxed.h` header file. If the user does not specify a memory consistency mode the relaxed mode will be used by

default. Memory consistency modes will be further explained in section 2.3. For more about the header files please refer to appendix A. Lines 4 to 7 are similar to C, starting with main, and a simple C print statement. `THREADS` and `MYTHREAD` are special values that will be explained in 2.2.1 and 2.2.2 respectively.

2.2 Most Commonly Used Keywords and Functions

To quickly understand and help make the transition from C to UPC, this section will examine some of the most commonly used keywords and functions. It is followed by a basic UPC program in section 2.5.

2.2.1 THREADS

The keyword `THREADS` signifies the number of threads that the current execution is utilizing. The value of `THREADS` can be defined either at compile time or at runtime. To define `THREADS` at compile time, the user simply sets its value along with the compiling options. To define `THREADS` at runtime, the user does not compile with a fixed number of threads, but instead specifies the number of threads in the run command. A common use for the keyword `THREADS` is to set up work sharing among the threads.

2.2.2 MYTHREAD

The keyword `MYTHREAD` is used to determine the thread number currently being executed. Example 2.2-1 shows how to use `MYTHREAD` to specify that only thread 0 perform an extra task while all the threads print the simple "`Hello World`" statement:

Example 2.2-1:

```
1:     #include <upc_relaxed.h>
2:     #include <stdio.h>
3:     void main(){
4:          if (MYTHREAD==0){
5:               printf("Rcv'd: 'Starting Execution' from THREAD %d\n",
6:               MYTHREAD );
7:          }
```

```
8:
9:          printf("Hello World from THREAD %d (of %d THREADS)\n",
10:         MYTHREAD, THREADS);
11:     }
```

This example is similar to example 2.1-1, with the exception of lines 4-7. The `if` condition in line 4 specifies the thread number that should execute the statements that follow the condition. Thus only thread 0 will execute lines 5 and 6. The remaining lines, lines 9 to 10, are executed by every thread.

2.2.3 upc_forall

The `upc_forall()` is a work sharing construct that looks similar to a `for` loop with the exception of having a fourth parameter, which determines the thread that should run the current iteration of the loop.

```
upc_forall ( expression; expression; expression; affinity)
```

This fourth parameter, also known as the affinity field, accepts either an integer which is translated to (integer `% THREADS`); or an address which is used to determine the thread to which the address has its affinity. Iterations of a `upc_forall` must be independent of one another. Here is a quick example of `upc_forall`:

```
1:     upc_forall(i=0; i<N; i++; i){
2:          printf("THREAD %d (of %d THREADS) performing iteration %d\n",
3:          MYTHREAD, THREADS, i);
4:     }
```

The fourth parameter of the `upc_forall` call, or more specifically `i % THREADS,` controls which thread will be running the ith loop iteration. It is equivalent to saying:

```
1:     for(i=0;i<N;i++)
2:       if(MYTHREAD==i%THREADS)
3:          printf("THREAD %d (of %d THREADS) performing iteration %d\n",
4:                MYTHREAD, THREADS, i);
```

If instead an address is provided for the affinity field, the thread with the affinity to the address will iterate through the current loop. Example 2.2-2 provides an example of how a user would specify an address instead of an integer for the fourth parameter of a upc_forall statement.

Example 2.2-2:

```
1:    #include <upc_relaxed.h>
2:    #include <stdio.h>
3:    #define N 10
4:    shared [2] int arr[10];
5:
6:    int main(){
7:            int i=0;
8:            upc_forall (i=0; i<N; i++; &arr[i]){
9:                    printf("THREAD %d (of %d THREADS) performing iteration
%d\n",
10:                   MYTHREAD, THREADS, i);
11:           }
12:
13:           return 0;
14:   }
```

In line 3 the size of N is defined to be 10. In line 4 the manner in which the array `arr` is distributed across the threads is determined, namely in a round robin fashion with a chunk size of 2. In line 8 the fourth parameter of the upc_forall statement is the address `&arr[i].` At runtime this is replaced with the value of the thread which has affinity to `arr[i].` This `upc_forall` statement is essentially equivalent to:

Example 2.2-3:

```
1:          for (i=0; i<N; i++){
2:                if (MYTHREAD== upc_threadof(&arr[i])){
3:                      printf("THREAD %d (of %d THREADS) performing
iteration %d\n",
4:                      MYTHREAD, THREADS, i);
5:                }
6:          }
```

2.2.4 shared

UPC's distributed shared memory model provides the user with two memory spaces, the private and the shared. The private memory is accessed similar to the way one would in C. To use the shared memory space however, the user will need to use the `shared` qualifier. There are a few ways to use this `shared` qualifier:

```
1:     int local_counter;                // private variable
2:     shared int global_counter;        // shared variable
3:     shared int array1[N];             // shared array
4:     shared [N/THREADS] int array2[N]; // shared array
```

```
5:      shared [ ] int array3[N];          // shared array
6:      shared int *ptr_a;                 // private pointer to shared
7:      shared int *shared ptr_c;          // shared pointer to shared
```

In line 1 `local_counter` is defined we as normally do in C. Thus, it is a private variable. Every thread would have a private local copy of that variable. In contrast, in line 2 `global_counter` is declared as `shared int`. It resides in thread 0's shared memory and every thread will have access to it. In general, the declaration:

```
shared [block_size] type variable_name
```

means that the variable is shared with a layout qualifier of "`block_size`". Thus the shared object is distributed across the entire shared memory space in the span of block size per thread. If the block size is not provided, the default block size of 1 will be applied.

In line 3, `array1` of size `N` is defined using the shared qualifier and no layout is given. This translates to `shared [1] int array1[N]`, and thus the array will be distributed across the shared memory in a round robin fashion with a block_size 1, as depicted in figure 2.2-1.

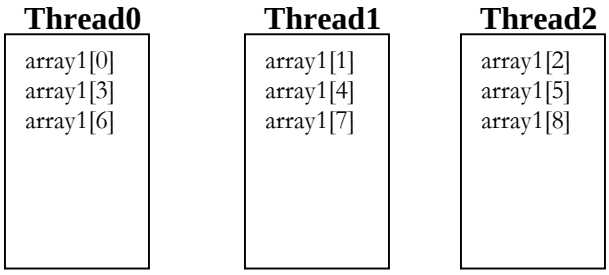


Figure 2.2-1 Using the default block_size, ie. 1, array `array1[9]` would be distributed as shown.

In line 4, `array2` of size `N` is defined using the shared qualifier with a block_size of `N/THREADS`. This translates to the distribution of `array2` in chunks of (`N/THREADS`) to each thread in a round robin manner. Since the block_size is an integer, if `N` is not fully divisible by `THREADS`, the floor of the division will be taken as shown in Figure 2.2-2.

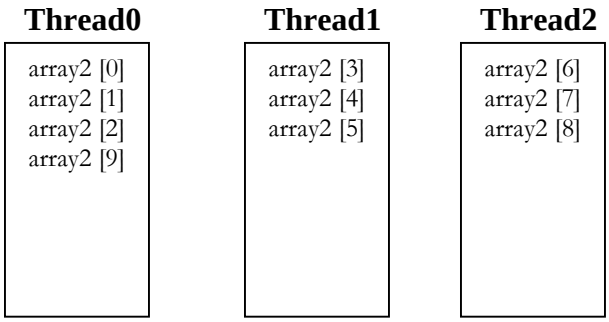


Figure 2.2-2 Using the block_size `N/THREADS`, where `N=10`, and THREADS=3, array `array2[N]` would be distributed as shown.

In line 5, `array3` of size `N` is defined using the shared qualifier with a block_size of `[ ]`. In this case the entire array is placed into thread0, as shown in figure 2.2-3.

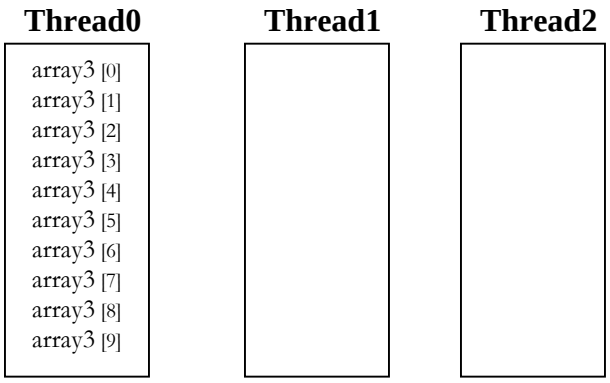


Figure 2.2-3 Using an infinite block_size `[]`, array3`[N]`would be completely reside in thread 0.

Lines 6-7 show examples of how users may use pointers with shared qualifiers. More details on data and pointer arithmetic in UPC can be found in chapter 3.

2.2.5 upc_barrier

The `upc_barrier` statement is commonly used to synchronize all threads before any of the threads continue. This is generally used when a data dependency occurs between the threads. To use `upc_barrier` simply place it at your desired synchronization point. Example 2.2-4 is an example showing how to use barriers:

Example 2.2-4:

```
1:    #include <upc_relaxed.h>
2:    #include <stdio.h>
3:
4:    shared int a=0;
5:    int b;
6:
7:    int computation(int temp){
8:            return temp+5;
9:    }
10:
11:   int main(){
12:           int result=0, i=0;
13:           do {
14:                   if (MYTHREAD==0){
15:                           result = computation(a);
16:                           a = result*THREADS;
17:                   }
18:                   upc_barrier;
19:                   b=a;
20:                   printf("THREAD %d: b = %d\n", MYTHREAD, b);
21:                   i++;
22:           }       while (i<4);
23:           return 0;
24:   }
```

The main function is defined in lines 12 to 23. Thread 0 computes the value of a in lines 14 to 17 of the `do-while` loop. A `upc_barrier` is placed in line 16 to ensure that all the threads wait until the value of `a` is set before carrying on. In line 19 each thread assigns the value of the shared variable a (on thread 0) to its private variable b. If a `upc_barrier` is not placed in line 18, there is no guarantee that all the threads are updating `b` using the latest value of `a`. Details about synchronization can be found in chapter 5.

2.2.6 upc_lock/upc_unlock

A common way to ensure that the shared element is not accessed by other threads while it is being updated by one thread is to use lock statements. This prevents other threads from reading or modifying the object until the lock has been released. UPC provides lock and unlock mechanisms via:

```
upc_lock(upc_lock_t *ptr);
upc_unlock(upc_lock_t *ptr);
```

Example 2.2-5 is an example that shows the usage of upc_lock/upc_unlock:

Example 2.2-5:

```
1:     #include <upc_relaxed.h>
2:     #include <stdio.h>
3:     #include <math.h>
4:
5:     #define N 1000
6:
7:     shared [] int arr[THREADS];
8:     upc_lock_t *lock;
9:     int main (){
10:            int i=0;
11:            int index;
12:
13:            srand(MYTHREAD);
14:            if ((lock=upc_all_lock_alloc())==NULL)
15:               upc_global_exit(1);
16:            upc_forall( i=0; i<N; i++; i){
17:                    index = rand()%THREADS;
18:                    upc_lock(lock);
19:                    arr[index]+=1;
20:                    upc_unlock(lock);
21:            }
22:            upc_barrier;
23:            if( MYTHREAD==0 ) {
24:                    for(i=0; i<THREADS; i++)
25:                            printf("TH%2d: # of arr is %d\n",i,arr[i]);
26:                    upc_lock_free(lock);
27:            }
28:
29:            return 0;
30:    }
```

In line 7 an array `arr` of size `THREADS` is declared with `[]` layout qualifier signifying that the entire array will reside in thread 0. In line 8 `lock` is defined as a pointer to type `upc_lock_t`. The memory is allocated for `lock` in line 14 using `upc_all_lock_alloc()`. The call to srand in line 13 makes rand in line 17 give different results on different threads. There is contention for access to the `arr` array as all the threads try to update array entries as seen in line 19. Thus, the access to the shared array `arr` is controlled using `upc_lock()` in line 18 and released using `upc_unlock()` in line 20. The `upc_barrier` in line 22 is crucial to ensure that all the threads are done with their updating. In lines 23 to 25 thread 0 simply print out the results. And finally in line 26 `upc_lock_free()` is used to release the memory used by `lock`.

2.3 Memory Consistency

When data is shared among several threads, memory consistency becomes an issue, and UPC provides several ways to ensure memory consistency. The first method is to enforce strict data consistency, where shared data is synchronized each time before access. This implies that if the shared data is currently being updated by another thread, the thread will wait for a synchronization point before accessing the data. Strict mode also prevents the compiler from rearranging the sequence of independent shared access operations for optimizations. This can result in significant overhead and should be avoided if possible. The second mode is the relaxed mode, where the threads are free to access the shared data any time. This mode is the default mode because it allows the compiler to freely optimize the code to achieve better performance. Here are the ways in which the memory consistency modes may be defined in UPC:

In a global scope:

```
#include <upc_strict.h>
#include <upc_relaxed.h>
```

When defined in a global scope any shared object that is not specifically defined as strict or relaxed will take on the mode specified by the global definition.

In a sectional scope:

```
#pragma upc strict
#pragma upc relaxed
```

The `#pragma upc [strict|relaxed]` allows the user to specify the mode of synchronization within a block of code. `#pragma upc [strict|relaxed]` will be the mode until the end of the block. This overrides the global scope.

At the variable level

```
strict shared [N/THREADS] array1[N]
relaxed shared [] array2[10]
```

At the variable level, the defined mode will override any other default, global, or sectional mode specified. This is especially useful if there are various modes used in the code such as those defined by `#pragma upc [strict|relaxed]` and the need to enforce a mode on an object is critical.

2.4 Compiling and Running UPC Programs

To compile and run your UPC program, it is best to refer to the compiler manual of your specific machine. Appendix B gives some of the commands for some implementations. In general you would compile using a UPC compiler, which takes a number of options; one of them can be the number of threads. In general, to compile a UPC code most compilers adopt similar compile time parameters:

```
<UPC compile command> <thread options> <optimizations> <code> –o <output>
```

for example the compile command for HP UPC would look something like this:

```
upc –fthreads 4 -02 helloworld.c –o helloworld
```

Here upc is the compile command, `-fthreads` is the thread option, `-02` specifies the optimization level desired, `helloworld.c` tells the compiler the program name (most compilers support either `.c` or `.upc` extensions), and finally `–o helloworld` specifies the executable's output name.

Running a UPC program also varies from compiler to compiler. Some compilers require the use of a special keyword while others allow you to run the program as if it were a regular executable. In general, to run a UPC executable most compilers adopt similar runtime parameters:

```
<upc runtime command> <thread option> <executable>
```

for example the runtime command for HP UPC would look something like this:

```
prun –n 4 helloworld
```

`Here prun` is the UPC runtime command, `-n 4` is the thread option specifying how many threads to run with, and `helloworld` specifies the executable's name. Specifying the number of threads is not necessary if the number of threads has already been specified at compile time.

The advantage of being able to dynamically assign the number of threads at run time is that the same executable will be able to run on any number of processes without requiring a recompile. However, specifying the number of threads at compile time allows better compiler optimizations and fewer restrictions on how the code could be written.

2.5 Vector Addition Example

Having covered the basic features and most commonly used UPC keywords and functions, it is time to take a look at a complete UPC program. Vector Addition is a basic example that has been chosen to highlight some of the basic concepts of UPC. As the name implies, vector addition performs addition of two vectors and places the result in a third vector:

Example 2.5-1:

```
1:     #include<upc_relaxed.h>
2:     #define N 100
3:     shared int v1[N], v2[N], v1plusv2[N];
4:     void main()
5:     {
6:        int i;
7:        for(i=0;i<N;i++)
8:           if(MYTHREAD==i%THREADS)
9:              v1plusv2[i]=v1[i]+v2[i] ;
10:
11:    }
```

The code is noticeably similar to a typical C code with the exception of a few UPC specific qualifiers and keywords. In line 1 the inclusion of `upc_relaxed.h` signifies that this code will not follow the strict memory consistency model and will allow the compiler to optimize the order of shared accesses for the best performance. Continuing on to line 3, the shared qualifier signifies that the variables will be shared among the threads, and since there is no block_size specified it will be distributed in a round robin manner across the threads until all data elements are exhausted.

Assuming 4 threads, vectors `v1`, `v2`, and `v1plusv2` will be distributed among the different threads as seen in figure 2.5-1.

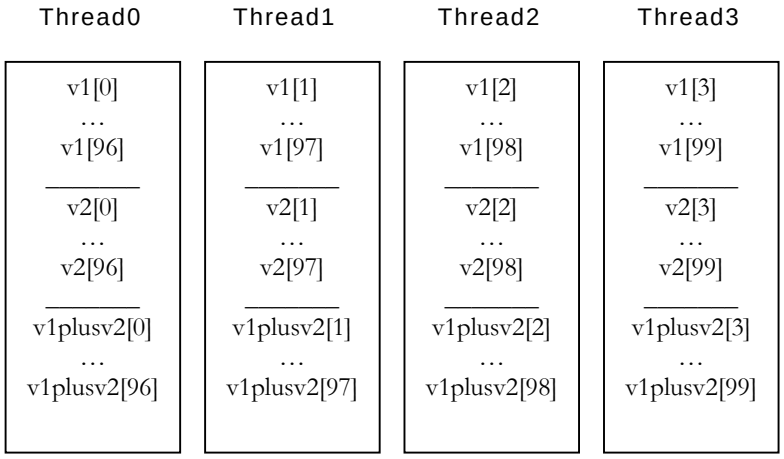


Figure 2.5-1 Default Distribution of Shared Array Elements

In line 8, `MYTHREAD` determines which thread is going to execute the current iteration. Each thread will calculate the modulo division given by `i%THREADS` which produces values that range from 0 to `THREADS-1`. A thread executes the current iteration i only if the value of `i%THREADS` is equal to its thread number, `MYTHREAD`.

UPC provides a simpler way to perform the same loop through the use of the `upc_forall` statement.

Example 2.5-2:

```
1:     #include<upc_relaxed.h>
2:     #define N 100
3:     shared int v1[N], v2[N], v1plusv2[N];
4:     void main()
5:     {
6:        int i;
7:        upc_forall(i=0;i<N;i++;i)
8:              v1plusv2[i]=v1[i]+v2[i] ;
9:
10:    }
```

The `upc_forall` statement in line 7 is a simpler way to define work sharing among the threads. The difference between a normal C for loop and the `upc_forall` loop is the fourth field, called the

affinity field. The affinity field determines which thread will execute the current iteration of the loop body in a `upc_forall`. As a result the conditional statement that was used in example 2.5-1 is removed and `for` replaced by `upc_forall`. The previous two examples work well with such work distributions since all iterations are independent.

In example 2.5-2, iteration i will be executed by thread `i%THREADS`. Given the round robin default distribution of the elements of the arrays, all computations in this example will be local and require no remote memory accesses. The affinity field of the `upc_forall` can also be a shared reference. If instead of `i`, the affinity field was a reference to the shared memory space, the loop body of the `upc_forall` statement is executed by the thread which hosts this shared address; see Example 2.2-2.

Chapter

3

3 Data and Pointers in UPC

This chapter examines data and pointers in UPC. Data distribution, pointers and pointer arithmetic, as well as special pointer functions are covered.

3.1 Shared and Private Data

UPC differentiates between two different kinds of data, shared and private. When declaring a private object, UPC allocates memory for such an object in the private memory space of each thread. Thus, multiple instances, one per thread, of such an object will exist. On the other hand, shared objects are allocated in the shared memory space.

Example 3.1-1 shows a UPC code for declaration of different identifiers. Figure 3.1-1 shows the corresponding distribution, assuming 5 threads.

Example 3.1-1:

```
1:     shared int x[12];
2:     int y;
3:     shared int z;
```

In line 1 of example 3.1-1 x is a shared array that will be distributed across the shared memory space in a round robin fashion. This will result in the distribution of the elements `x[0]`, `x[1]`,..`x[4]` across threads 0, 1,..,4, respectively. Elements `x[5]`, `x[7]`,..,`x[11]` will wrap around, thus, spreading

across threads 0, 1,..,4, as shown in figure 3.1-1. Since y is defined as a scalar private variable in line 2, UPC will allocate memory for the variable y on all the available threads. Finally, the variable `z` is declared as scalar shared variable and, therefore, UPC will allocate memory space for `z` only on one thread, thread 0.

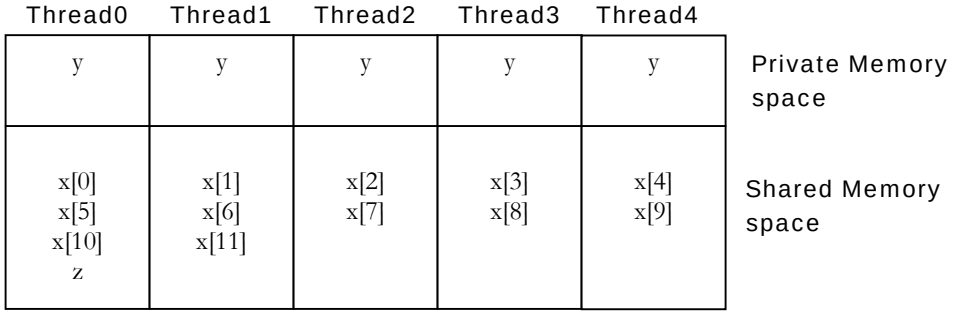


Figure 3.1-1 Affinity Example

3.2 Blocking of Shared Arrays

Because the default element-by-element round robin distribution of data may not fit the requirements of many applications, UPC allows block distribution of data across the shared memory space. UPC users can declare arbitrary blocking sizes to distribute shared arrays in a block per thread basis as follows:

```
shared [block-size] array [number-of-elements];
```

When the `[block-size]` is omitted, the default block size of `1` is assumed.

Example 3.2-1:

```
shared[3] int x[12];
```

In example, 3.2-1, `x` is a shared array that will be distributed, according to the `[3]` layout qualifier, across the shared memory space in 3-element blocks per thread in a round robin fashion. Assuming the number of threads is 3, then elements `x[0]`, `x[1]`, and `x[2]` will have affinity to thread 0; `x[3]`, `x[4]`, and, `x[5]` will have affinity to thread 1, and so on, as seen in figure 3.2-1.

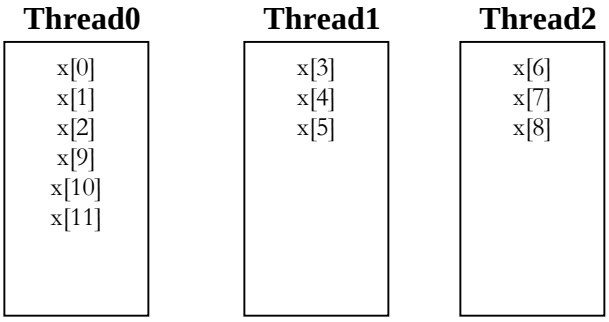


Figure 3.2-1 Distributing blocked one-dimensional arrays

The following example shows how UPC distributes two dimensional arrays with user specified block_size.

```
shared [2] int A[4][2];
```

Here, the important facts are the block_size and the fact that C stores two dimensional arrays in a row-major order. Assuming 3 threads, this declaration will result into the data layout seen in figure 3.2-2.

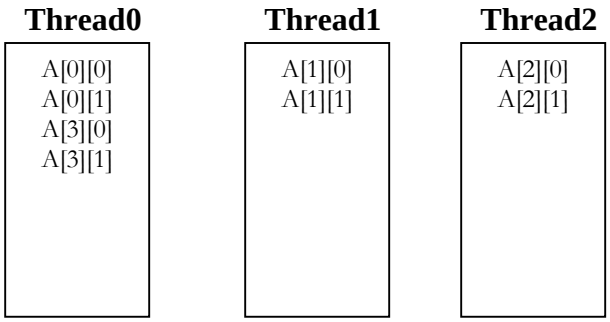


Figure 3.2-2 Distributing blocked two-dimensional arrays

Here is another example showing the usage of block_size in shared arrays:

Example 3.2-3: Matrix by Vector Multiply

```
1:     #include<upc_relaxed.h>
2:     #define N 100*THREADS
3:     shared [N] double A[N][N];
4:     shared double b[N], x[N];
5:     void main()
6:     {
7:          int i,j;
8:          /* reading the elements of matrix A and the
9:          vector x and initializing the vector b to zeros
```

```
10:         */
11:         upc_forall(i=0;i<N;i++;i)
12:               for(j=0;j<N;j++)
13:                     b[i]+=A[i][j]*x[j] ;
14:    }
```

In this example, line 3 distributes matrix `A` in the shared space one row per thread in a round robin fashion. In line 4, the `b` and `x` vectors have a one element per thread distribution. See example 4.1-1 for further discussion of this code.

3.3 UPC Shared and Private Pointers

To understand UPC pointers, it is important to understand both where the pointer points and where it resides. There are four distinct possibilities: private pointers pointing to the private space, private pointers pointing to the shared space, shared pointers pointing to the shared space, and lastly shared pointers pointing to the private space. These four cases are shown in figure 3.3.1.

		Where does the pointers reside?	
		Private	**Shared**
Where does the pointer point?	**Private**	**PP**	**PS**
	Shared	**SP**	**SS**

Figure 3.3-1 UPC Pointer Possibilities

Declaring a UPC pointer is similar to the way a pointer in C is declared. Here is a look at the options provided in figure 3.3-1:

Example 3.3-1:

```
1:     int *p1;               // private to private
2:     shared int *p2;        // private to shared
3:     int *shared p3;        // shared to private (not advised)
4:     shared int *shared p4; // shared to shared
```

Line 1 declares a pointer `p1`, which points to a private space and resides in the private space, a standard C pointer. In line 2 `p2` is declared as a private pointer that points to the shared space, and thus each thread has an instance of `p2`. Such a pointer can be advantageous in speed and flexibility. In line 4, `p4` is a shared pointer pointing to the shared space. There is only one instance of `p4`, residing on thread 0. Finally in line 3, `p3` is defined as a shared pointer to the private space. This type of declaration will create an access violation for any thread that is not the owner of that private space and therefore it should be avoided.

3.4 Shared and Private Pointer address format

Unlike ordinary C pointers, a UPC pointer-to-shared has to keep track of a number of things. These are the thread number; the virtual address of the block, and the phase that indicates to which item in the block the pointer is pointing.

An example implementation of a pointer-to-shared is shown in 3.4-1. Pointers to shared objects, in this case, have three fields: the thread number, the local address of the block, and the phase (which specifies position in the block). This is graphically sketched in figure 3.4-1 with the number of bits that represents each part of the pointer.

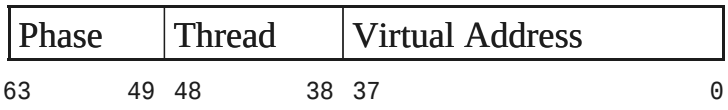


Figure 3.4-1 An example implementation of a UPC pointer format

3.5 Special UPC Pointer Functions and Operators

There are a number of special functions that can access the values of the different fields inside pointer-to-shared address representations. One is `upc_threadof (shared void *ptr)`, which returns the number of the thread that has affinity to the shared object pointed to by `ptr`. Another is

`upc_phaseof (shared void *ptr)`, which returns the position within the block of the pointer `ptr`. The third is `upc_addrfield(shared void *ptr)` which returns the address of the block which is pointed at by the pointer-to-shared `ptr`.

There are also a number of associated operators with the UPC pointers. The `upc_localsizeof (type-name or expression)` operator returns the size of the local portion of a shared object. The `upc_blocksizeof (type-name or expression)` operator returns the block_size associated with operand. And `upc_elemsizeof (type-name or expression)` operator returns the size (in bytes) of the left-most type that is not an array.

3.6 Casting of Shared to Private Pointers

Pointers-to-shared may be cast to private pointers only if the shared data has affinity to the current thread. The result will be a valid private pointer referring to the same data. This operation is useful if there is a large block of shared data on the current thread that may be more efficiently manipulated through the private pointer. Note that a local pointer cannot be cast to a pointer-to-shared.

Example 3.6-1: Casting of Shared Pointer to a Private Pointer

```
1:     shared int x[THREADS];
2:     int *p;
3:     p=(int *)&x[MYTHREAD];
```

In this example, each of the private pointers will point to the x element, which has affinity to its own thread, i.e. `MYTHREAD`.

Example 3.6-2: The Role of Casting of Shared Pointers

```
1:     shared [3] int *p;
2:     shared [5] int *q;
3:     p=q;
```

The last assignment statement is acceptable; however, some implementations may require an explicit type cast. Despite being assigned to `q`, pointer `p` will obey pointer arithmetic for block_size of 3 and not 5.

3.7 UPC Pointer Arithmetic

The block_size determines where a pointer-to-shared points to and has affinity to when incremented or decremented.

Example 3.7-1: UPC Pointer Arithmetic

Assuming we have 4 threads

```
1:     #define N 16
2:     shared int x[N];
3:     shared int *p=&x[5];
4:     p=p+3;
```

In line 3 the pointer `p` initially points to data, `x[5]` , which has affinity with thread 1. After incrementing `p` in line 4, `p` points to data `x[8]` , which has affinity with thread 0 as seen in figure 3.7-1, where the dotted arrow denotes the position of `p` before incrementing.

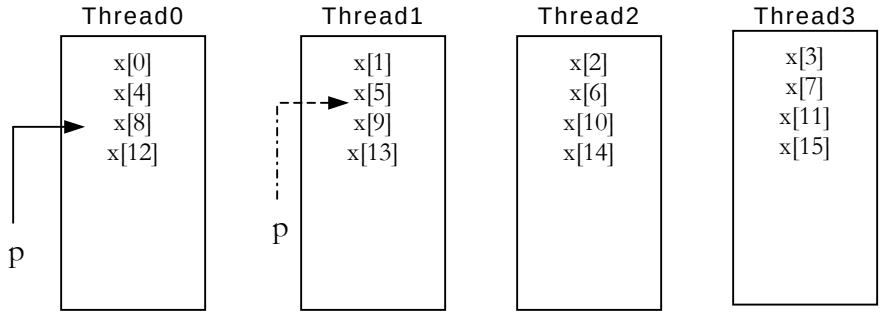


Figure 3.7-1 Pointer Casting Example

Example 3.7-2: How Shared Pointer Arithmetic Follows Blocking.

```
1:     #define N 16
2:     shared [3] int x[N];
3:     shared int *p=&x[4];
4:     p=p+7;
```

In this case `x` has a block_size of 3, and the shared pointer `p` will be pointing to `x[4],` which is in thread 1. After incrementing, `p` will be pointing at `x[12]` on thread 0 as seen in figure 3.7-2.

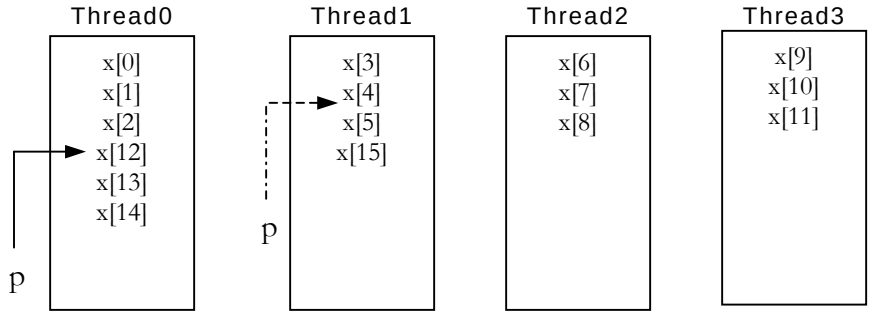


Figure 3.7-2 Blocking and Pointers

3.8 UPC String Handling Functions

UPC has library functions to copy data to and from shared memory, analogous to the C `memcpy()`. Three functions are provided, depending on whether source or destination is in shared or private memory.

`upc_memcpy( dst, src, n)` copies shared to shared memory.
`upc_memput( dst, src, n)` copies from private to shared memory.
`upc_memget( dst, src, n)` copies from shared to private memory.

The function `upc_memcpy( dst, src, n)` copies `n` bytes from a shared object (`src`) with affinity to one thread, to a shared object (`dst`) with affinity to the same or another thread. (Neither thread needs to be the calling thread.) It treats the `dst` and `src` pointers as if they had type: `shared [ ] char [n]`. The effect is equivalent to copying the entire contents of one shared array object with this type (`src`) to another (`dst`).

The function `upc_memput( dst, src, n)` copies `n` bytes from a private object (`src`) on the calling thread, to a shared object (`dst`) with affinity to any single thread. It treats the `dst` pointer as if it had type: `shared [ ] char [n]`. The effect is equivalent to copying the entire contents of a private array object(`src`) declared as `char[n]` to a shared array object (`dst`) of the above shared type.

The function `upc_memget (dst, src, n)` copies n bytes from a shared object (`src`) with affinity to any single thread, to a private object (`dst`) on the calling thread. It treats the `src` pointer as if it

had type: `shared [ ] char [n]`. The effect is equivalent to copying the object(`dst`) declared as `char[n]`.

In each example below we assume that `THREADS` is specified at compile time and that `SIZE` is a multiple of `THREADS`.

Example 3.8-1: (Assuming that SIZE%THREADS == 0)

```
1:     #include <upc_relaxed.h>
2:     #define SIZE 16000
3:
4:     shared int data[SIZE];
5:     shared [] int th0_data[SIZE];
6:
7:     int main()
8:     {
9:             int i, sum;
10:            sum = 0;
11:            if (MYTHREAD==0){
12:               for( i=0; i<THREADS; i++ )
13:                    upc_memcpy(&th0_data[i*(SIZE/THREADS)],       &data[i],
(SIZE/THREADS)*sizeof(int));
14:               for( i=0; i<SIZE; i++ )
15:                    sum += th0_data[i];
16:            }
17:            return 0;
18:    }
```

In Example 3.8-1 lines 4 and 5 declare two shared arrays. The array `data` is distributed across all the threads using the default block_size of 1, while `th0_data` resides solely on thread 0 by declaring the infinite blocking factor. In lines 11 to 16, thread 0 copies the contents of the shared array `data` to its private shared array `th0_data` using multiple `upc_memcpy` calls, each `upc_memcpy` copying the part of `data[] that` has affinity with thread `i`. That is, `data[i]`, `data[i+THREADS]`, `data[i+2*THREADS]` and so forth. In lines 14 and 15 thread 0 calculates the global sum of the data array without any remote accesses.

The second function `upc_memput (dst, src, size)` copies from private to shared memory space:

Example 3.8-2:

```
1:     #include <upc_relaxed.h>
2:     #define SIZE 16000
3:
4:     shared int data[SIZE];
5:
6:     int main()
7:     {
8:             int i, sum;
9:             int localbuf[SIZE/THREADS];
10:            sum = MYTHREAD;
11:            for( i=0; i<SIZE/THREADS; i++ ) {
12:                    localbuf[i] = sum;
13:                    sum += i*THREADS;
14:            }
15:            upc_memput(&data[MYTHREAD],localbuf,     (SIZE/THREADS)*sizeof(
int));
16:            return 0;
17:    }
```

In Example 3.8-2 line 4 declares a shared array `data` of size `SIZE,` which is distributed across all the threads using the default block_size of 1. In lines 11 to 15, each thread initializes its local array `localbuf` and then copies the `localbuf` contents to shared array `data` using `upc_memput`.

And finally the third function `upc_memget (dst, src, size)` copies from shared to private memory space:

Example 3.8-3:

```
1:     #include <upc_relaxed.h>
2:     #define SIZE 10000
3:
4:     shared int data[SIZE];
5:
6:     int main()
7:     {
8:             int i, sum;
9:             int localbuf[SIZE];
10:            for( i=0; i<THREADS; i++ )
11:                    upc_memget(&localbuf[i*(SIZE/THREADS)],
&data[MYTHREAD], (SIZE/THREADS)* sizeof(int));
12:            sum = 0;
13:            for( i=0; i<SIZE; i++ )
14:                    sum += localbuf[i];
15:            return 0;
16:    }
```

Example 3.8-3 is similar to example 3.8-1 with the exception that the destination array resides in private space.

Chapter

4

4 Work Distribution

This chapter considers the methods in which work can be equally distributed among threads.

4.1 Data Distribution for Work Sharing

As explained in chapter 3, array data distribution among threads in UPC can be done in the declaration statements. Take for example:

```
1:     #include <upc_relaxed.h>
2:     #define IMG_SIZE 512
3:     shared [(IMG_SIZE*IMG_SIZE)/THREADS] int image[IMG_SIZE][IMG_SIZE];
```

In line 3 the entire image is broken into `THREADS` blocks, i.e. one block per thread. If `THREADS` were 8, the block_size would be 32768.

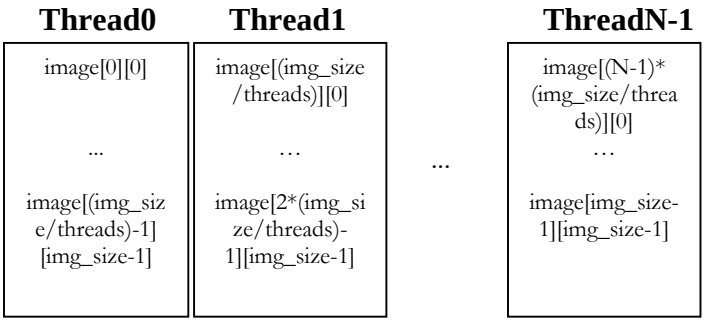


Figure 4.1-1 Distributing Data across the Threads

Data distribution and work sharing in UPC are the core for exploiting data locality. Work sharing capabilities in UPC are provided by the upc_forall construct for distributing independent iterations across the threads. For a quick overview of `upc_forall` please refer to section 2.2.3. Here we revisit

the Matrix by Vector Multiply example from chapter 3 showing the usage of `upc_forall` to distribute the workload equally among all the threads:

Example 4.1-1: Matrix by Vector Multiply

```
1:     #include<upc_relaxed.h>
2:     #define N 100*THREADS
3:     shared [N] double A[N][N];
4:     shared double b[N], x[N];
5:     void main()
6:     {
7:          int i,j;
8:          /* reading the elements of matrix A and the
9:          vector x and initializing the vector b to zeros
10:         */
11:         upc_forall(i=0;i<N;i++;i)
12:               for(j=0;j<N;j++)
13:                     b[i]+=A[i][j]*x[j] ;
14:    }
```

Line 3 distributes the matrix `A` one row per thread in a round robin fashion. In line 4 the `b` and `x` vectors have the default block_size of 1, so they are distributed round-robin on element per thread. Thus row `i` of matrix `A` and element `i` of vectors `b` and `x` all have the same affinity, to thread (`i%THREADS`). The work is done by the `upc_forall` loop in line 11. This has the affinity parameter `i`, which distributes the iterations across the threads in round-robin order, with iteration `i` being executed by thread (`i%THREADS`). Because of the way shared data is distributed, the iterations are independent, and both `A[i][j]` and `b[i]` are local data in iteration `i`.

An equivalent way to specify the work is to replace line 11 by:

```
11:   upc_forall(i=0; i<N; i++; &A[i][0])
or
11:   upc_forall(i=0; i<N; i++; A[i])
```

This uses the affinity of row `i` of the matrix `A` to determine the thread that does the iteration. In this case it is just thread (`i%THREADS`) as before.

Chapter

5

5 Synchronization and Memory Consistency

UPC allows for several different synchronization mechanisms such as: barriers, locks, and memory consistency control. UPC barriers can ensure that all threads reach a given point before any of them can proceed any further. Locks are needed to coordinate access to critical sections of the code. Finally, memory consistency control gives the ability to control the access ordering to memory by the different threads such that performance can be maximized with no risk of data inconsistency.

5.1 Barrier Synchronization

Barriers are useful to ensure that all the threads reach the same execution point before any of them proceeds any further. UPC provides two basic kinds of barriers, blocking barriers and split-phase barriers (non-blocking). The blocking barrier is invoked by calling the function `upc_barrier` and the split-phase barrier is obtained by calling the function pair `upc_notify` and `upc_wait`. Figure 5.1-1 graphically explains the concept of a barrier.

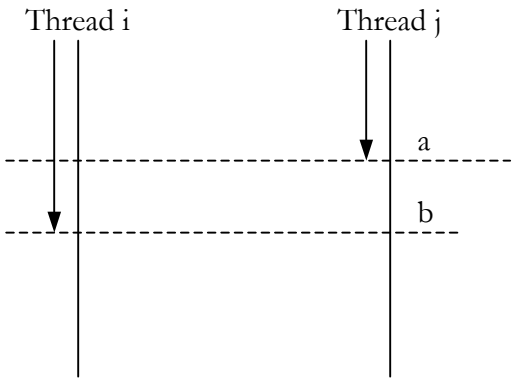
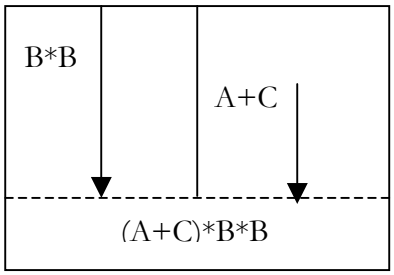


Figure 5.1-1 Synchronization Barriers

In case of blocking barriers, when thread i reaches point **b**, it will be blocked from further execution until thread j and all other threads reache the same point. In the case of non-blocking barriers a thread can continue local execution after notifying that it has reached point **a**. At point **a**, thread i executes a notify and at **b** it performs a wait. Thread i can perform any independent work between these two points while some of the other threads have not yet reached the barrier. When all other threads reach at least point **a** and execute a notify, thread i can proceed. Example 5.1-1 shows the use of the non-blocking barrier. The program is used to calculate (A+C)*B*B, where A, B, and C are all N*N matrices. This example assumes `THREADS` is defined at compiled time and `N` is a multiple of `THREADS`.

Example 5.1-1: Barrier Synchronization

```
1:     shared [N]int A[N][N];
2:     shared [N]int C[N][N];
3:     shared [N]int B[N][N];
4:     shared [N]int ACsum[N][N];
5:     shared [N]int Bsqr[N][N];
6:     shared [N]int Result[N][N];
7:
8:     void matrix_multiplication (shared[N] int result[N][N],
                                   shared[N] int m1[N][N],
                                   shared[N] int m2[N][N]){
9:         int i, j, l, sum;
10:        upc_forall(i=0;i<N;i++; &m1[i][0]){
11:            for(j=0;j<N;j++){
12:               sum=0;
13:                     for(l=0;l<N;l++)
14:                   sum+=m1[i][l]*m2[l][j];
15:               result [i][j]=sum;
16:            }
17:        }
18:    }
19:
20:    matrix_multiplication(Bsqr,B,B);
21:    upc_notify 1;
22:    upc_forall(i=0;i<N;i++;&A[i][0]){
23:        for(j=0;j<N;j++)
24:            ACsum[i][j]+=A[i][j]+C[i][j];
25:    }
26:    upc_wait 1;
27:    matrix_multiplication(Result, ACsum, Bsqr);
```



In this example B*B is first calculated and stored in the matrix `Bsqr` and matrices `A` and `C` are added together and stored in matrix `A`. Finally matrix `A` and `Bsqr` are multiplied and stored in the final matrix Result. However since the calculation of (B*B) and (A+C) are independent of each other there is no need to wait for (B*B) to finish computing before computing (A+C). This is where the use of a non-blocking barrier can provide more efficient performance. Let's take a closer look at the example. In the first few lines, lines 1-6, the shared structures are defined. The arrays `A`, `C`, `Bsqr`, and Result are all defined as shared arrays with a block distribution of `N`, that is, one row per thread, rows distributed round-robin among the threads. In the `matrix_multiplication` function, the `upc_forall` loop distributes the work among the threads, as seen in line 10, on a per row basis. The ith iteration is executed by the thread that has that row, i.e. `m[i][0]`. So on line 14 the row

`m1[i][l]` is local, but the column `m2[l][j]` is not. In line 20 (B*B) is calculated using the `matrix_multiplication` function. It is followed by a `upc_notify` in line 21 for the current thread to tell the other threads that it has completed calculating its rows of (B*B). Since `upc_notify` is not a blocking barrier mechanism, the call returns immediately and lines 22-25 are executed. In lines 22-25 the task (A+C) is distributed among the threads on a per row basis using the `upc_forall` loop, where the thread having affinity to `A[i][0]` executes the current loop. Once thread `i` has calculated the row of (A+C), it waits until all the other threads have executed the `upc_notify` in line 21. This means that all rows of (B*B) have been calculated, and it is safe for thread I to proceed to calculate its rows of the final product (A+C)*(B*B).

5.2 Synchronization Locks

In UPC, shared data can be protected against multiple writers through the use of locks. The two constructs `void upc_lock(upc_lock_t *l)` and `void upc_unlock(upc_lock_t *l)` allow locking and unlocking of shared data so that it can be accessed on a mutually exclusive basis. The function `int upc_lock_attempt(upc_lock_t *l)` returns 1 on successful locking and 0 otherwise. This may improve performance by avoiding busy waits when a lock is not available.

Locks are created in two ways- collectively or non-collectively. The collective function is

```
upc_lock_t * upc_all_lock_alloc(void)
```

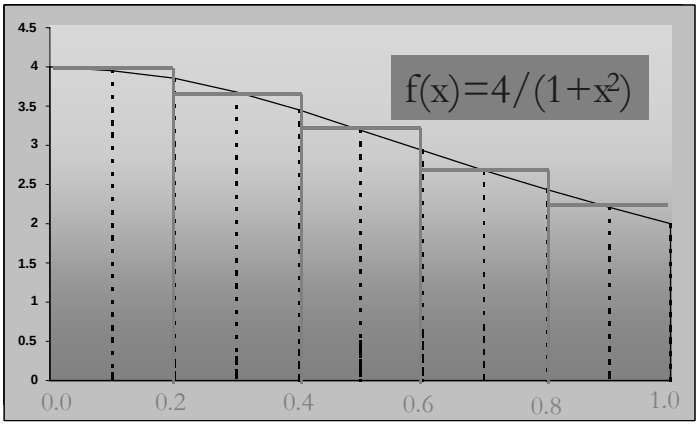
This function is called simultaneously by all threads. A single lock is allocated, in the unlocked state, and all threads receive a pointer to it. The non-collective function is

```
upc_lock_t * upc_global_lock_alloc(void)
```

This function is called by a single thread. A single lock is allocated, and only the calling thread receives a pointer to it, unless the returned thread lock pointer is stored in a shared variable. A subset of threads could use this lock to synchronize references to shared data. If multiple threads call this function, all threads which make the call get different allocations. (See chapter 6 for analogous functions `upc_all_lock_alloc` and `upc_global_lock_alloc` for allocating shared

memory.) The function upc_lock_free is called, by a single thread to free a lock allocated by either type of call.

Consider for example the computation of π using numerical integration, as shown in figure 5.2-1.



$$4\int_{1}^{0} \frac{1}{1+x^2}$$

Figure 5.2-1 Numerical Integration (computation of π)

Example 5.2-1: Numerical Integration with Locks

```
1:     //Numerical Integration
2:     //Example - The Famous PI
3:     #include<upc_relaxed.h>
4:     #include<math.h>
5:     #define N 1000000
6:     #define f(x) (1.0/(1.0+x*x))
7:     upc_lock_t *l;
8:     shared float pi = 0.0;
9:     void main(void)
10:    {
11:         float local_pi=0.0;
12:         int i;
13:         l=upc_all_lock_alloc();
14:
15:         upc_forall(i=0;i<N;i++; i)
16:               local_pi +=(float) f((.5+i)/(N));
17:         local_pi *= (float) (4.0 / N);
18:
19:         upc_lock(l);
20:         pi += local_pi;
21:         upc_unlock(l);
22:
23:         upc_barrier; // Ensure all is done
24:         if(MYTHREAD==0) printf("PI=%f\n",pi);
24:         if(MYTHREAD==0) upc_lock_free(l);
25:    }
```

In example 5.2-1 threads collectively call `upc_all_lock_alloc()` to create a memory for the lock l as shown in line 13. The task of calculating `pi` is divided among the threads using the `upc_forall`

loop in lines 15-16. Each thread accumulates its portion of the summation in its own private local_pi. In lines 19-21 the local sums are added to get the global answer. To prevent overlapping updates to `pi`, all threads call `upc_lock`. As each thread acquires the lock it adds its `local_pi` to `pi`, then releases the lock. Then it executes `upc_barrier` to wait for the other threads to complete the updates to `pi`. Finally, in line 24, only one thread calls `upc_lock_free`.

Example 5.2-2 is a generic example showing how the use of two different locks. Half the threads use lock `l1` to update `v1` and half use `l2` to update `v2`.

Example 5.2-2: More Locks

```
1:     //create the locks
2:     upc_lock_t *l1;
3:     upc_lock_t *l2;
4      shared float v1=1.0, v2=2.0;
5:     l1=upc_all_lock_alloc();
6:     l2=upc_all_lock_alloc();
7:
8:     if (MYTHREAD>THREADS/2) update_v1();
9:     else update_v2();
10:
11:    void update_v1()
12:    {
13:         upc_lock(l1);
14:         v1=expression1(v1);
15:         upc_unlock(l1);
16:    }
17:
18:    void update_v2()
19:    {
20:         upc_lock(l2);
21:         v2=expression2(v2);
22:         upc_unlock(l2);
23:    }
```

5.3 Ensuring Data Consistency

UPC provides three different ways to define the memory consistency mode. This section will describe several methods to go about ensuring that data consistency is met.

There are two consistency modes available to the user: strict and relaxed. The main difference between the two modes is that in the strict mode, synchronization is key to shared data access. Say

thread 3 is accessing shared data, and thread 2 needs to access the same shared data. In the strict mode, thread 2 will wait for thread 3 to finish accessing the data, and then proceed. In the relaxed mode however, thread 2 will have no knowledge that thread 3 is currently accessing the data and would start overwriting data written by thread 3. This is because unlike the strict mode, no implicit synchronization steps were taken prior to accessing the data in the relaxed mode; it is left up to the user to determine when to perform synchronization.

To define a memory consistency mode, the user can use any of the three scopes: define it at the program level, the block level, or the variable level. To define the consistency mode at the program level, the user simply uses one of two headers: `upc_strict.h` for defining a strict mode throughout the program:

```
1:     #include <upc_strict.h>
2:     void main(){
3:     }
4:     ...
```

Or `upc_relaxed.h` for defining a relaxed mode throughout the program:

```
1:     #include <upc_relaxed.h>
2:     void main(){
3:     }
4:     ...
```

The advantage of using the program level scope is the convenience of defining the consistency method at the beginning and not having to worry about it throughout the program. This however becomes overkill when defining the strict consistency mode. The difference in speed and performance between strict mode and relaxed mode is quite significant. In relaxed mode the compiler is able to optimize the memory accesses as it sees fit, since it assumes the ordering does not matter. However in strict mode, the compiler is told not to perform any optimizations, leaving the optimizations solely to the user. Synchronization prior to each access in the strict mode also presents significant overhead as compared to immediate access in the relaxed mode.

The second scope level is code block level scoping. This is used primarily to change the default mode, or override the mode defined at the program level for a particular section of the code. The user can do so by placing a pragma statement at the beginning of a compound-statement {…}. The mode and then reverts back to the original mode at the end of the block by using another.

```
1:     #include <upc_relaxed.h>
2:     shared int counter=0;
3:     void main(){
4:          ...
5:          {     //perform this block in strict mode
6:                #pragma upc strict
7:                counter++;
8:                printf("Counter now shows %d\n", counter);
9:          }
10:         //reverts back to relaxed mode
11:         ...
12:    }
```

An advantage for using the section level scope is to reduce the overhead faced when defining a particular scope at the program level. Users will most likely use this scope to enforce a strict mode of memory consistency. The disadvantage is that it is necessary to plan ahead and design sections of the program which would require strict access and others that do not and carefully set pragmas.

The third scope level is the object level. It is used primarily to override the default mode or program level mode for a shared variable. To do so the user simply prefixes the strict or relaxed keyword to the variable declaration.

```
1:     #include <upc_relaxed.h>
2:     strict shared int counter;
3:     void main(){
4:          counter++;
5:          printf("Counter now shows %d\n", counter);
6:          }
7:     }
```

An advantage of defining the consistency mode at the variable level is that it allows the user to enforce synchronization for that variable only. Variables that have their mode explicitly set are not affected

Example 5.3-1 shows how memory consistency works in UPC. Here, `flag_ready` is an important flag that needs to be set only if the statement above it has completed. To ensure this, the `strict` keyword is used upon defining the `flag_ready` variable.

Example 5.3-1 Memory Consistency Example

```
1:     strict shared int flag_ready = 0;
2:     shared int result0, result1;
3:
4:     switch(MYTHREAD) {
5:          case 0 :
6:               result0 = expression1;
7:               flag_ready=1; //if not strict, it could be
8:                             //switched with the above statement or
9:                             //executed concurrently
10:              break;
11:         case 1 :
12:              while(!flag_ready); //Same note
13:              result1=expression2+result0;
14:              break;
15:    }
```

Example 5.3-1 shows the subtle things to watch for when the ordering of statements is of key importance. Here, `flag_ready` is a trigger for the other threads to perform their tasks. In case 0, result0 must be assigned expression1 before setting the ready flag. Since the threads are run in parallel, the sequence in which the threads are run cannot be guaranteed, thus the moment when the other threads pick up the value of `result0`, `result1`, or `flag_ready` is unknown. To ensure proper setting and reading of values, the strict mode is necessary. Without the strict mode, it is quite possible that the ordering between `result0` and `flag_ready` assignments can be switched. Similarly in case 1, the result1 may be assigned before the expression `while(!flag_ready)` is executed. Declaring a variable as strict will enforce synchronization, and then only one thread can modify the variable at any given time. The other threads will have to wait their turn to read or modify the variable. In the `relaxed` mode however, the thread will have no knowledge of the accesses to the shared variable and thus will immediately access the variable, which may lead to memory inconsistency issues.

Chapter

6

6 Dynamic Memory Allocation in UPC

This chapter considers the methods in which shared memory is dynamically allocated by UPC. Since UPC is an extension to Standard C, the functions of Standard C can be used to allocate and free memory in private space.

6.1 Dynamic Shared Memory Allocation

Dynamic shared memory allocation in UPC can be collective or non-collective, global or local. There are four allocation functions:

```
upc_all_alloc
upc_global_alloc
upc_alloc
upc_local_alloc (deprecated in language specification V1.1)
```

The first two are similar:

```
shared void *upc_all_alloc (size_t nblocks, size_t nbytes)
shared void *upc_global_alloc (size_t nblocks, size_t nbytes)
```

Both functions are *global*: they allocate shared space across all threads, compatible with the declaration:

```
shared [nbytes] char[nblocks*nbytes]
```

`upc_all_alloc` is a collective function; that is, it must be called by all threads with the same arguments, and it returns the same pointer value on all threads. On the other hand, `upc_global_alloc` is not a collective function; it is called by one thread. If called by more than one thread, multiple regions are allocated, and each calling thread gets a pointer to its own allocation. Figures 6.1-1 and 6.1-2 make clear the similarities and differences between these two functions.

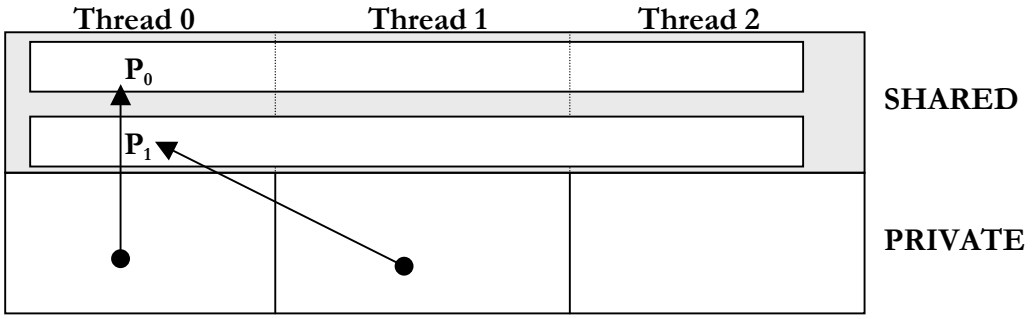


Figure 6.1-1 Using upc_global_alloc

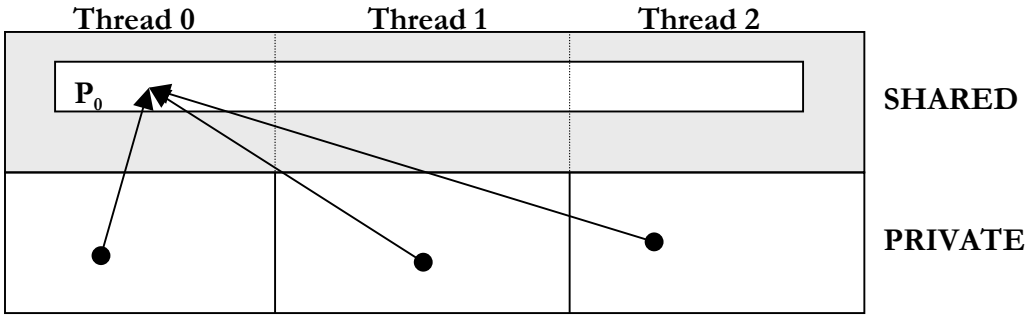


Figure 6.1-2 Using upc_all_alloc

The other two functions are *local*: they allocate shared memory that has affinity to the calling thread only. The function `upc_local_alloc` is in the original language specification, but is deprecated in specification V1.1, to be replaced by `upc_alloc`. Both functions have the same effect: `upc_alloc` allocates `nbytes` bytes, while `upc_local_alloc` allocates `nblocks*nbytes` bytes. The functions are similar to the declaration:

```
shared [] char [nbytes]
```

except that this allocates shared memory with affinity to thread 0 only.

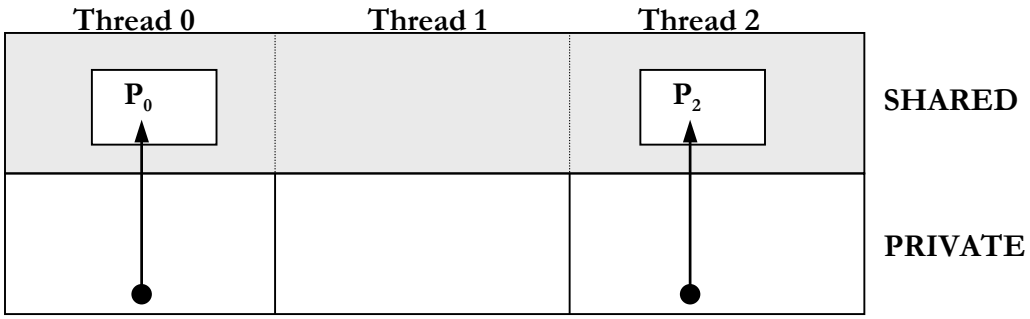


Figure 6.1-3 Using upc_alloc

6.2 Freeing Memory

The function

```
void upc_free (shared void *ptr)
```

frees the dynamically allocated shared storage pointed to by `ptr`. In the case of memory allocated by `upc_all_alloc`, any thread, may call `upc_free` to free the memory; but only one such call has any effect. Example 6.2-1 shows `upc_free` is used to free memory allocated by the `upc_all_alloc` function.

Example 6.2-1 Shared Memory Dynamic Allocation

```
1:     shared [10] int *table;
2:     int main()
3:     {
4:        /* allocate a buffer of 10*THREADS, with block_size of 10
      elements */
5:        table = (shared [10] int *)upc_all_alloc(THREADS, 10* sizeof(int));
6:
7:        /* do some work here */
8:        (…)
9:
10:       /* free the table, any thread may free table */
11:       if (MYTHREAD==0)
12:           upc_free( table );
13:    }
```

6.3 Memory Allocation Examples

The following example 6.3-1 illustrates different shared memory dynamic allocations:

Example 6.3-1 Different scenarios of Shared Memory Dynamic Allocations

```
0:     /* shared variable declarations */
1:     shared [5] int *p1, *p2, *p3;
2:     shared [5] int * shared p4, * shared p5;
3:
4:     /* Allocate 25 elements per thread, with each thread
5:        doing its portion of the allocation. – COLLECTIVE CALL */
6:     p1 = (shared [5] int *)upc_all_alloc(5*THREADS, 5*sizeof(int));
7:
8:     /* Allocate 25 elements per thread, but just run the
9:        allocation on thread 5. – NON COLLECTIVE CALL */
10:    if (MYTHREAD == 5)
11:      p2 = (shared [5] int *)upc_global_alloc(5*THREADS,
                                                 5*sizeof(int));
12:
13:    /* Allocate 5 elements only on thread 3. NON COLLECTIVE CALL */
14:    if (MYTHREAD == 3)
15:      p3 = (shared [5] int *)upc_alloc(sizeof(int)*5);
16:
17:    /* Allocate 25 elements per thread, just run the allocation
18:     on thread 4, but have the result be visible everywhere.- NON-
        COLLECTIVE CALL */
19:    if (MYTHREAD == 4)
20:      p4 = (shared [5] int shared *)upc_global_alloc(5*THREADS,
                                                  5*sizeof(int));
21:
22:    /* Allocate 5 elements only on thread 2, but have the
23:       result visible on all threads. */
24:    if (MYTHREAD == 2)
25:      p5 = (shared [5] int shared *)upc_alloc(sizeof(int)*5);
26:
27:    /* De-allocate p1, any thread may free p1*/
28:    if( MYTHREAD == 0 )
29:        upc_free( p1 );
30:
31:    /* De-allocate p2, only thread 5 may free p2*/
32:    if( MYTHREAD == 5 )
33:         upc_free( p2 );
34:
35:    /* De-allocate p3*/
36:    if( MYTHREAD == 3 )
37:         upc_free( p3 );
38:
39:    /* De-allocate p4 & p5 */
40:    if( MYTHREAD == 0 ) {
41:         upc_free( p4 );
42:         upc_free( p5 );
43:    }
```

Chapter 7

7 UPC Optimization

There are several ways to enhance the performance of UPC through either compiler and runtime optimizations and/or hand–tuning. These are discussed in this chapter along with specific examples.

7.1 How to Exploit the Opportunities for Performance Enhancement

Performance optimizations are typically possible through:

- Compiler optimizations
- Run-time system
- Hand tuning

7.2 Compiler and Runtime Optimizations

An advanced programmer should become familiar with the UPC specific compiler optimization options. The user should also be aware of whether the vendor has a run time system that can help optimize your code and how to set respective environment variables. When everything else fails, the following hand tuning techniques can be used.

7.3 List of Hand Tunings for UPC Code Optimization

The performance of UPC code can be improved using the following hand tuning methods:

1. Use local pointers instead of shared pointers when dealing with local shared data, through casting and assignments

2. Use block copy instead of copying elements one by one with a loop

3. Overlap remote accesses with local processing using split-phase barriers

7.3.1 Using local pointers instead of shared pointers

UPC compilers may generate code which takes longer to access local shared data than private data. Thus, for better performance all UPC local shared accesses must be turned into UPC private accesses. This step is called privatization.

Example 7.3-1 illustrates how to privatize local shared accesses in a UPC code, or in other words, how to convert UPC local shared accesses to UPC private accesses to obtain an effective memory bandwidth.

Example 7.3-1 Privatization example

```
1:     int *pa, *pc;
2:     upc_forall(i=0;i<N;i++;&A[i][0]) {
3:          *pa = (int*) &A[i][0];
4:          *pc = (int*) &C[i][0];
5:          for(j=0;j<P;j++)
6:                pa[j]+=pc[j];
7:     }
```

Pointer arithmetic is typically faster using private pointers than shared pointers. In some cases, pointer de-referencing can be an order of magnitude faster.

7.3.2 Aggregation of Accesses Using Block Copy

When UPC shared remote accesses are needed, aggregating such accesses and fetching them as a block is better than multiple reads/writes since latency and other overheads only appear once.

Example 7.3-2 shows how to use a block copy instead of a standard single element copy. The block copy is done using a string function, very similar to the ones found in the C language.

Example 7.3-2 Block copy by string function copy example

Instead of

```
1:     shared [] int a[1000], b[1000];
2:     // Copy element by element
3:     for (j=0; j<1000; j++)
4:       b[j]=a[j];
```

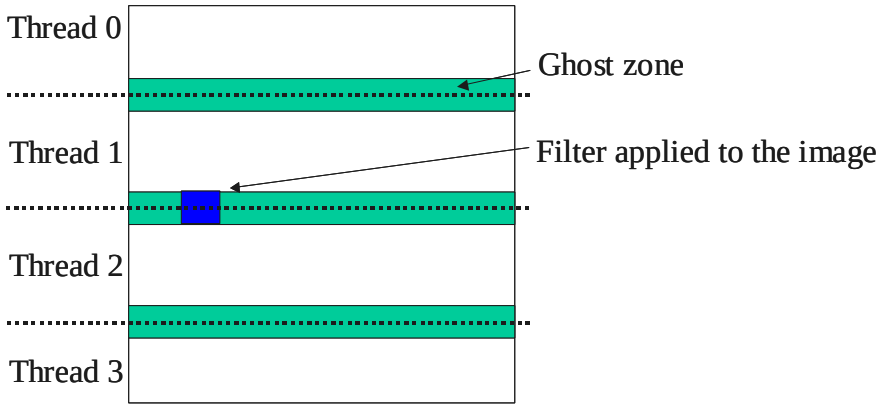
```
1:     // Copy the whole array at once using string functions
2:     upc_memcpy(b, a , sizeof(a));
```

7.3.3 Overlapping Remote Accesses with Local Processing

In order to hide the time spent in remote shared accesses, it is possible to overlap communication with computations. This is can be done using split-phase barriers instead of blocking barriers. In this case local processing can be done while waiting for data or synchronization.

The example 7.3-3 shows a brief implementation of computation and communication overlapping. Ghost zones are prefetched and while waiting for this prefetching, the computation can be done on all the local shared data except for the ghost zones. After completion of such processing, upc_wait() waits for all threads to complete the communication step, and thereafter the ghost zone can be processed.

Example 7.3-3 Overlapping and split-phase barriers



```
1:     upc_memcpy(ghost_copy, ghost_zone, size);
2:     upc_notify;
3:     // work on everything but the ghost_zones
4:     upc_wait;
5:     // work with the ghost_zones
```

Chapter

8

8 UPC Programming Examples

This chapter provides two detailed examples, Sobel Edge and the N Queens. These examples were chosen to highlight some of the features of UPC.

8.1 Sobel Edge Detection

Original Image

Edge-detected Image

Figure 8.1-1: Edge Detection

Edge detection, figure 8.1-1, has many applications in computer vision, including image registration and image compression. One popular way of performing edge detection is using the Sobel operators. The process involves the use of two masks, see figure 8.1-3, for detecting horizontal and vertical edges through convolution with the underlying image.

In parallelizing this application, the image is partitioned into equal contiguous slices of rows, as seen in figure 8.1-2, which are distributed across the threads, as blocks of a shared array. With such

contiguous horizontal distribution, remote accesses into the next thread only will be needed when the mask is shifted over the last row of a thread data to access the elements of the next row.

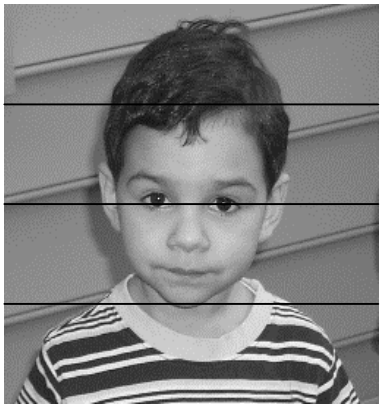


Figure 8.1-2: Image distribution across 4 threads

Sobel Edge images are essentially images that are processed through a convolution applying the horizontal edge and vertical edge detection masks. Each pixel is evaluated using a template, consisting of eight of its neighboring pixels, which is applied using the two masks. The results from the masks are added together to determine the edge value for that pixel, as shown in figure 8.1-3.

Template

10	20	25	20	15	10	10	20
80	100	90	95	105	100	105	110
40	30	35	15	20	25	80	40
20	20	30	60	80	100	200	40
10	40	45	50	60	70	205	40
40	45	30	80	60	80	230	50
60	100	110	110	80	80	255	50
40	30	25	10	10	10	200	50

Image

- 1	0	1
- 2	0	2
- 1	0	1

West Mask
Used to detect vertical edge

- 1	- 2	- 1
0	0	0
1	2	1

North Mask
Used to detect horizontal edge

Figure 8.1-3: Sobel Edge Illustrated

To see how the Sobel Edge is applied, example 8.1-1 shows a serial C code that applies the Sobel Edge on an image stored in the `orig[N][N]` array and stores the result into the `edge[N][N]` array.

Example 8.1-2: Sobel Edge Detection Written in C

```
1:     #define BYTE unsigned char
2:     BYTE orig[N][N],edge[N][N];
3:     int Sobel(){
4:         int i,j,d1,d2;
5:             double magnitude;
6:             for (i=1; i<N-1; i++){
7:                for (j=1; j<N-1; j++){
8:                    d1 = (int) orig[i-1][j+1] - orig[i-1][j-1];
9:                    d1 += ((int) orig[i][j+1] - orig[i][j-1]) << 1;
10:                   d1 += (int) orig[i+1][j+1] - orig[i+1][j-1];
11:                   d2 = (int) orig[i-1][j-1] - orig[i+1][j-1];
12:                   d2 += ((int) orig[i-1][j] - orig[i+1][j]) << 1;
13:                   d2 += (int) orig[i-1][j+1] - orig[i+1][j+1];
14:                   magnitude = sqrt(d1*d1+d2*d2);
15:                   edge[i][j]= magnitude>255? 255:(BYTE)magnitude;
16:               }
17:           }
18:        return 0;
19:    }
```

In a serial C lines 7 to 17 apply the edge detection algorithm to each square of 9 pixels. Lines 8 to 10 apply the west mask, while lines 11 to 13 apply the north mask. In lines 14 to 15, the magnitude is then calculated and stored as the edge. This is repeated for every pixel of the image (except those around the corner).

To distribute the work into parallel tasks using UPC, the image first is subdivided into a number of blocks, and the work is distributed among the threads using `upc_forall`. The threads then compute the convolution of each pixel in parallel providing the edge image.

Example 8.1-2 is shows how Sobel Edge Detection could be written in UPC .

Example 8.1-2:

```
1:     #define BYTE unsigned char
2:     shared [N*N/THREADS] BYTE orig[N][N],edge[N][N];
3:     int Sobel(){
4:         int i,j,d1,d2;
5:         double magnitude;
6:         upc_forall (i=1; i<N-1; i++; &edge[i][0]){
```

```
7:              for (j=1; j<N-1; j++){
8:                 d1 = (int) orig[i-1][j+1] - orig[i-1][j-1];
9:                 d1 += ((int) orig[i][j+1] - orig[i][j-1]) << 1;
10:                d1 += (int) orig[i+1][j+1] - orig[i+1][j-1];
11:                d2 = (int) orig[i-1][j-1] - orig[i+1][j-1];
12:                d2 += ((int) orig[i-1][j] - orig[i+1][j]) << 1;
13:                d2 += (int) orig[i-1][j+1] - orig[i+1][j+1];
14:                magnitude = sqrt(d1*d1+d2*d2);
15:                edge[i][j] = magnitude>255? 255 : (BYTE)magnitude;
16:             }
17:          }
18:          return 0;
19:    }
```

Only a few minor changes to the sequential C code were needed to turn it into UPC. In line 2 the arrays `orig` and `edge` are declared as shared. The image is distributed equally among the threads using a block size of `N*N/THREADS` (assuming `N` is a multiple of `THREADS`). Thus each thread gets a chunk of `N/THREADS` rows. In line 6 the C `for` loop is changed to a `upc_forall` loop to distribute the workload. For each iteration the thread with affinity to the `i`th row, `edge[i]`, will execute the code. Only the first row and the last row of pixels generated in each block need remote memory reading.

8.2 N-Queens

In the N Queens problem we seek to find all solutions to the problem of placing N queens on an NxN chessboard such that no queen can kill another. This means that no two queens may be placed on the same row, column, or diagonal. The algorithm uses depth-first searching and backtracking.

As the problem size increases so does the number of iterations that will be required to search all possible ways that the N queens can co-exist on the same board.

The parallel solution to this problem is very straightforward, because in this tree search algorithm, branches of the tree are totally independent. In order to reduce the granularity of the jobs and increase their number, for increased scalability, a job is described as searching along one of the subtrees that correspond to a given row-column position combination in the first L rows. All threads proceed to perform the sequential search along their own subtrees. Figure 8.2-1 gives an

example of work distribution where each job is based on the position combination of the first two rows. The remote accesses associated with this algorithm are minimal and the parallel algorithm is therefore embarrassingly parallel.

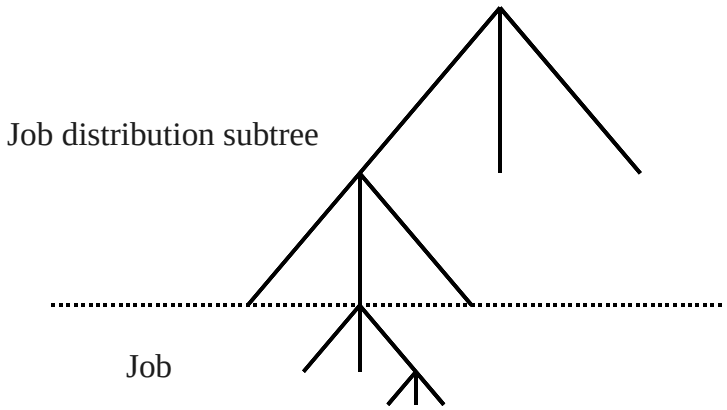


Figure 8.2-1 Job Distribution Tree

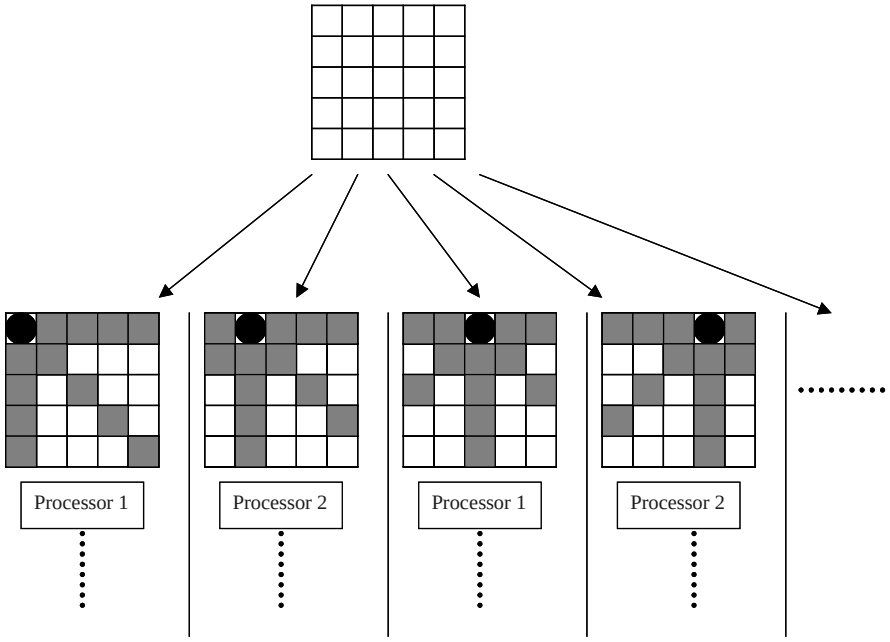


Figure 8.2-2 Nqueens Problem using UPC

The tree search is broken into subtrees, where each branch represents an independent thread performing the sequential search, Figure 8.2-2.

Example 8.2-1 shows how Nqueens could be written in UPC.

Example 8.2-1: Nqueens in UPC

```
1:     //Main program - variables
2:
3:     shared int number_solns[THREADS];
4:     // parameters
5:     shared int n;          // Problem size
6:     shared int l;          // Distribution
7:     shared int method;     // Round-robin / chunking
...
8:     //Main program - initialization
9:
10:     if (MYTHREAD==0) {
11:         n=atoi(argv[1]);
12:         if ((n<=0) || (n>16)) {
13:         fprintf(stderr,"0<n<17\n");
14:         upc_global_exit(0);
15:         }
16:         l=atoi(argv[2]);
17:         if ((l<0) || (l>=n)) {
18:         fprintf(stderr,"0<=l<n\n");
19:         upc_global_exit(0);
20:         }
21:    }
...
22:    //Main program - execution
23:
24:    upc_barrier; // make sure thread 0 has set the parameters
25:    number_solns[MYTHREAD] = sched(n,l,method);
26:    upc_barrier; // Complete all solutions before reduction
27:    nsols=0;
28:    if (MYTHREAD==0) {
29:         for(i=0;i<THREADS;i++)
30:               nsols+=number_solns[i];
31:    }
...
32:    //Code for job distribution
33:    int sched(int n, int l, int method) {
34:         // Distribution in a round-robin fashion:
35:         if (method==roundrobin)
36:         upc_forall(j=0;j<njobs;j++; j) {
37:               call sequential algorithm
38:         }
39:         if (method==chunk)
40:         // or Distribution in a chunking fashion:
41:         upc_forall(j=0;j<njobs;j++; (j*THREADS)/njobs ) {
42:               call sequential algorithm
43:         }
44:    }
```

The line numbers are numbered sequentially for ease of viewing, however please be aware that the "..." signifies code in between that was not included for the sake of brevity. At line 3, an array of size `THREADS` is created to store the number of solutions each thread discovers. This is declared as `shared int number_solns[THREADS]` with a default distribution of `[1]`, i.e. each thread has affinity to its solution count index of the array. All the threads wait for the initialization to complete by using a barrier in line 24. The solutions are then calculated using the inline function `sched`, shown in lines 33-44. `Sched` sets the job distribution strategy as chunks or as simple round robin. Another barrier is placed in line 26 to ensure that all the threads are done finding all possible solutions. In lines 28 to 31 thread 0 aggregates the number of solutions.

Appendix

A

Appendix A: Programmer's Reference Guide

Reserved Words in UPC

In addition to the reserved words and functions that C utilizes, here is a list of keywords and functions that a UPC compiler will recognize:

Keywords:

```
MYTHREAD
relaxed
shared
strict
THREADS
UPC_MAX_BLOCK_SIZE
```

Functions:

```
upc_addr_field
upc_all_lock_alloc
upc_global_lock_alloc
upc_all_alloc
upc_alloc
upc_barrier
upc_blocksizeof
upc_elemsizeof
upc_fence
upc_forall
upc_free
upc_global_alloc
upc_global_exit
upc_localsizeof
upc_lock_attempt
upc_lock_free
upc_lock_t
upc_memcpy
upc_memget
upc_memput
upc_memset
upc_notify
upc_phaseof
upc_threadof
upc_resetphase
upc_unlock
upc_wait
```

The usage of these keywords and functions is examined throughout this document. The semantics, synopsis, and detailed description of these keywords and functions can be found in the UPC Specifications document [ElG03].

Libraries and Headers

UPC is built as an extension to the C language rather than a whole new language. This allows UPC to leverage the capabilities of C, and augment parallel capabilities.

There are three standard headers in UPC. They are:

```
<upc_strict.h>
<upc_relaxed.h>
<upc.h>
```

Specifying `upc_strict.h` notifies the compiler that "strict" mode will be used shared data accesses will be synchronized across all threads. This means that if any concurrent write accesses to the shared data are being performed, the strict mode will enforce a sequential resolution of writes. This is in contrast to upc_relaxed.h where no synchronization is enforced and each thread is free to access the shared data at will. It is important to note however that unless concurrent access to shared data is being performed, `upc_relaxed` and `upc_strict` will behave similar to each other.

The header `upc.h` defines common definitions, parameters, and utilities used provided by UPC. Both `upc_strict.h` and `upc_relaxed.h` include `upc.h`. However if only `upc.h` is included the synchronization mode will default to relaxed.

UPC Keywords

`THREADS`: Total number of threads
`MYTHREAD`: Identification number of the current thread (between `0` and `THREADS-1`)
`UPC_MAX_BLOCK_SIZE`: Maximum block size allowed by the compilation environment

Shared variable declaration

Shared objects

Shared variables are declared using the type qualifier "`shared`". Shared objects have to be declared statically (that is, either as global variables or with the keyword `static`).

Example of shared object declaration:

```
shared int i;
shared int b[100*THREADS];
```

The following will not compile if you do not specify the number of threads:

```
shared int a[100];
```

All the elements of `a` in thread 0:

```
shared [] int a[100];
```

Assume `THREADS` is specified and `N` is a multiple of `THREADS`. To distribute elements round robin:

```
shared a[N][N]
shared [1] a[N][N]
```

To distribute rows of matrix round-robin:

```
shared [N] a[N][N]
```

To distribute one block of `N/THREADS` rows per thread:

```
shared[N*N/THREADS] a[N][N]
```

Shared pointers

Pointer to shared:

```
shared int* p;
```

Shared pointer to shared data:

```
shared int* shared sp;
```

Work sharing

Distributes the iterations in a round-robin fashion with wrapping from the last thread to the first thread:

```
upc_forall (i=0; i<N; i++; i)
```

Distribute the iterations by consecutive chunks:

```
upc_forall (i=0; i<N; i++; i*THREADS/N)
```

The iteration distribution follows the distribution layout of `a`:

```
upc_forall (i=0; i<N; i++; &a[i])
```

Synchronization

Memory consistency

Defines strict or relaxed consistency model for the whole program.

```
#include “upc_strict.h”
```
or
```
include “upc_relaxed.h”
```

Sets strict memory consistency for the rest of the file:

```
#pragma upc strict
```

Sets relaxed memory consistency for the rest of the file:

```
#pragma upc relaxed
```

All accesses to i will be done with the relaxed consistency model:

```
shared relaxed int i;
```
or
```
relaxed shared int i;
```

All accesses to i will be done with the strict consistency model:

```
strict shared int i;
```

Synchronize locally the shared memory accesses; it is equivalent to a null strict reference.

```
upc_fence;
```

Barriers

Synchronize the program globally:

```
upc_barrier [value];
```

Where `value` is an integer.

Non-blocking split phase barrier:

```
upc_notify [value];

// Non-synchronized statements
relative to this on-going barrier
...

upc_wait [value];
```

UPC operators

`upc_threadof(p)` : thread having affinity to the location pointed by p
`upc_phaseof(p)` : phase associated with the location pointed by p
`upc_addrfield(p)` : address field associated with the location pointed by p
`upc_resetphase(p)` : reset phase associated with the location pointed by p

Dynamic memory allocation

Three different memory allocation methods are provided by UPC:

`upc_alloc (n)`: allocates n bytes of shared data in the calling thread only. It is called by one thread only.

`upc_global_alloc (n, b)`: globally allocates nxb bytes of shared data distributed across the threads with a block size of b bytes. It is called by one thread only.

`upc_all_alloc (n, b)`: collectively allocates nxb bytes of shared data distributed across the threads with a block size of b bytes. It needs to be called by all the threads.

`upc_free (p)`:Frees shared memory pointed to by p.

String functions in UPC

Equivalent of memcpy :

`upc_memcpy (dst, src, size)` : copy from shared to shared
`upc_memput (dst, src, size)` : copy from private to shared
`upc_memget (dst, src, size)` : copy from shared to private

Equivalent of memset:

`upc_memset(dst, char, size)` : initialize shared memory with a character

Locks

`upc_lock_t *l` : UPC lock type
`upc_all_lock_alloc()` : collectively returns pointer to dynamic lock
`upc_lock(l)` : blocking call, which waits for lock to become available
`upc_lock_attempt(l)` : grabs lock if available, returns lock status
`upc_unlock(l)` : releases lock
`upc_lock_free(l)` : releases memory allocated for lock

General utilities

Terminate the UPC program with exit status:

```
upc_global_exit(status);
```

Appendix

B

Appendix B: Running UPC on implementations

Available Compilers

Hardware Platform	Compiler Version
Cray T3D/E, Cray X1	UPC Compiler version 3.1.9
HP Alpha Server SC Family	UPC Compiler version 2.0, 2.1
SGI Origin Family	GCC-UPC version 3.2
IA32 Clusters	Michigan Tech MuPC 1.0 beta
IA32 and IA64 Clusters	UC Berkeley BUPC 1.0 beta
SUN Enterprise Server	SUN UPC Compiler "Beta"

Table B-1 Hardware Platform and Compiler Information

Table B-1 outlines a number of open source and vendor compilers that can be obtained and be used now, some with special arrangements. Ongoing and future implementations for IBM and Beowulf Cluster platforms are also underway.

Compiling and Running on Cray T3E

To compile with a fixed number (4) of threads:

```
upc –O2 –fthreads-4 –o vect_add vect_add.c
```

To run the program:

```
./vect_add
```

Compiling and Running on HP/Compaq

To compile with a fixed number of threads and run:

```
upc –O2 –fthreads 4 –o vect_add vect_add.c
prun ./vect_add
```

To compile without specifying the number of threads and run:

```
upc –O2 –o vect_add vect_add.c
prun –n 4 ./vect_add
```

Compiling and Running on SGI

To compile with a fixed number of threads and run:

```
upc -x upc -fupc-threads-4 –O2 -o vect_add vect_add.c
./vect_add
```

To compile without specifying the number of threads and run:

```
upc –x upc –O2 -o vect_add vect_add.c
./vect_add -fupc-threads-4
```

Compiling and Running on Berkeley UPC

To compile and run:

```
upcc vect_add.c –o vect_add
upcrun –n 4 ./vect_add
```

Appendix

C

Appendix C: Performance Tuning

Due to the number of compilers available to us and the number of unique customizations and/or tunings available to each compiler, it would be not feasible to mention them all in this document. However, as a sample case, this document will discuss some of the customizations and tunings possible for the HP UPC Compiler.

Runtime optimizations on HP (1)

Caching: The runtime system can cache the shared data to eliminate some remote fetches.

List of the related environment variables and (their default values):

```
–UPCRTS_USE_CACHE (false)
–UPCRTS_CACHE_SETS (128)
–UPCRTS_CACHE_BLOCK_SIZE (64)
–UPCRTS_CACHE_ASSOCIATIVITY (4)
–UPCRTS_DISP_CACHE_STATISTICS (False)
```

Runtime optimizations on HP (2)

Pre-fetching: The runtime system can pre-fetch the shared data to eliminate some remote fetches.

List of the related environment variables and (their default values):

```
–UPCRTS_USE_PREFETCH (False)
–UPCRTS_PREFETCH_DISTANCE (3)
```

```
–UPCRTS_PREFETCH_TRAINING (20)
–UPCRTS_DISP_PREFETCH_STATISTICS (False)
```

How to set an environment variable

With sh-like shell:

```
export UPCRTS_USE_CACHE=value
```

With csh shell:

```
setenv UPCRTS_USE_CACHE value
```

SMP local optimization

Allows threads to access the shared memory of any other thread in the same node as private. Compilation option (-smp_local). The advantage of using SMP local is that it eliminates communication overhead when accessing shared memory of other threads in the same SMP node.

National Security Agency (NSA - http://www.nsa.gov)

IDA Institute for Defense Analyses, Center for Computing Sciences (IDA - CCS http://www.super.org)

The George Washington University, High Performance Computing Laboratory (GWU HPCL - http://upc.gwu.edu)

Arctic Region Supercomputing Center (ARSC - http://www.arsc.edu)

The Hewlett-Packard Company (HP - http://h30097.www3.hp.com/upc/)

Cray Inc. (http://www.cray.com)

Etnus LLC. (http://www.etnus.com)

IBM (http://www.ibm.com)

Intrepid Technologies (http://www.intrepid.com)

Ernest Orlando Lawrence Berkeley National Laboratory (LBNL - http://upc.lbl.gov)

Lawrence Livermore National Laboratory (LLNL - http://www.llnl.gov)

Michigan Technological University (MTU - http://upc.mtu.edu)

Silicon Graphics, Inc. (SGI - http://www.sgi.com)

Sun Microsystems (http://www.sun.com)

University of California, Berkeley (http://www.berkeley.edu)

US Department of Energy (DoE - http://www.energy.gov)

Ohio State University (OSU - http://www.osu.edu)

Argonne National Laboratory (ANL - http://www.anl.ogv)

Sandia National Laboratory (http://www.sandia.gov)

University of North Carolina (UNC - http://www.unc.edu)

[Bro95] 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.

[Car99] William W. Carlson, Jesse M. Draper. Introduction to UPC and language specification CCS-TR-99-157.

[Cul93] 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.

[ElG03a] Tarek A. El-Ghazawi, William W. Carlson, Jesse M. Draper. UPC Language Specifications V1.1 (http://upc.gwu.edu). March, 2003.

[ElG03b] Tarek A. El-Ghazawi, François Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross, Dan Bonachea. UPC-IO: A Parallel I/O API for UPC
V1.0 pre9 (http://upc.gwu.edu). May, 2003.

[Eli03] Elizabeth Wiebel, David Greenberg, Steven Seidel. UPC Collective Operations Specification pre4V1.0 (http://upc.gwu.edu). April, 2003.

[MPI2] MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, July 18, 1997.

Index

affinity, 6, 7, 20, 23, 26, 27, 28
affinity field, 21
ANSI C, 5
barriers, 34
  blocking barriers, 35
  non-blocking barriers, 35
  split-phase, 48
block copy. *See* UPC Strings
block distribution, 23
blocking factor
  round robin, 21
blocking size, 24
casting, 48
data
  private data, 22
  shared data, 22
Data Coherency, 16
data distribution, 7, 23
  round robin, 23, 25
environment variables, 66
Examples
  Edge detection, 51
  Histogram, 51
  N Queens, 51, 54
  Sobel Edge, 51, 52
  Vector Example, 19
  Vector Multiply, 33
functions, 10, 58
hand–tuning, 47
headers, 59
  standard headers, 59
keywords, 10, 58
libraries, 59
locks, 34, 63
memory
  distributed shared memory, 5
  dynamic memory, 7
  global memory, 5
  memory model, 6
  shared memory, 22, 23
  static memory, 7
memory access
  remote memory access, 6, 21
memory allocation, 22, 62
  upc_all_alloc, 62
  upc_free, 62
  upc_global_alloc, 62
  upc_local_alloc, 62
memory consistency, 7, 8, 16, 34, 39, 61
  element scope, 40
  relaxed mode, 17, 39, 61
  section scope, 17, 40
  strict mode, 39, 61
  ucp_relaxed, 59
  upc_relaxed, 19
  upc_strict, 59
  using pragmas, 17
memory consistency scope
  element scope, 17
  program level scope, 17, 40
memory space
  private space, 6, 7, 25, 26
  shared space, 7, 25, 26
MYTHREAD, 10, 27, 59
optimizations
  compiler optimizations, 47
  hand tuning, 47
  runtime optimizations, 47
Performance optimizations, 47
phase, 26
pointer
  shared pointer, 7, 26, 28
Pointer address format, 26
pointers, 25
prefetching, 49
reserved words, 58
shared, 6, 12
  shared qualifier, 60
shared address, 21
split-phase, 34
THREADS, 10, 20, 21, 59
UPC, 5
UPC Compiler
  Alpha, 64
  Cray, 64
  GCC, 64
  SUN, 64
UPC Specifications V1.0, 5
UPC Strings, 62
  upc_memcpy, 62
  upc_memget, 31, 62
  upc_memput, 30, 62
  upc_memset, 62
upc_addrfield, 27, 62
upc_all_alloc, 43
upc_barrier, 14, 34, 61
upc_blocksizeof, 27
upc_elemsizeof, 27
upc_fence, 61
upc_forall, 11, 20, 21, 32, 51, 53, 60, 61
upc_global_exit, 63
upc_localsizeof, 27
upc_lock, 15, 37
upc_lock_attempt, 37
upc_lock_t, 16
upc_notify, 34, 62
upc_phaseof, 27, 62
upc_threadof, 26, 62
upc_unlock, 15, 37
upc_wait, 34, 49, 62

work sharing, 32, 60