UPC

DISTRIBUTED SHARED MEMORY PROGRAMMING

Tarek El-Ghazawi
The George Washington University

William Carlson
IDA Center for Computing Sciences

Thomas Sterling
California Institute of Technology

Katherine Yelick
University of California at Berkeley

A JOHN WILEY & SONS, INC., PUBLICATION

UPC

WILEY SERIES ON PARALLEL AND DISTRIBUTED COMPUTING

Series Editor: Albert Y. Zomaya

Parallel and Distributed Simulation Systems / Richard Fujimoto

Mobile Processing in Distributed and Open Environments / Peter Sapaty

Introduction to Parallel Algorithms / C. Xavier and S. S. Iyengar

Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences / Albert Y. Zomaya, Fikret Ercal, and Stephan Olariu (*Editors*)

Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches / Claudia Leopold

Fundamentals of Distributed Object Systems: A CORBA Perspective / Zahir Tari and Omran Bukhres

Pipelined Processor Farms: Structured Design for Embedded Parallel Systems / Martin Fleury and Andrew Downton

Handbook of Wireless Networks and Mobile Computing / Ivan Stojmenović (*Editor*)

Internet-Based Workflow Management: Toward a Semantic Web / Dan C. Marinescu

Parallel Computing on Heterogeneous Networks / Alexey L. Lastovetsky

Performance Evaluation and Characterization of Parallel and Distributed Computing Tools / Salim Hariri and Manish Parashar

Distributed Computing: Fundamentals, Simulations and Advanced Topics, *Second Edition* / Hagit Attiya and Jennifer Welch

Smart Environments: Technology, Protocols, and Applications / Diane Cook and Sajal Das

Fundamentals of Computer Organization and Architecutre / Mostafa Abd-El-Barr and Hesham El-Rewini

Advanced Computer Architecture and Parallel Processing / Hesham El-Rewini and Mostafa Abd-El-Barr

UPC: Distributed Shared Memory Programming / Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick

UPC

DISTRIBUTED SHARED MEMORY PROGRAMMING

Tarek El-Ghazawi
The George Washington University

William Carlson
IDA Center for Computing Sciences

Thomas Sterling
California Institute of Technology

Katherine Yelick
University of California at Berkeley

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

UPC : distributed shared memory programming / Tarek El-Ghazawi ... [et al.]
p. cm.
Includes bibliographical references and index.
ISBN-13 978-0-471-22048-0 (cloth)
ISBN-10 0-471-22048-5 (cloth)
1. UPC (Computer program language) 2. Parallel programming (Computer science) 3. Electronic data processing - - Distributed processing. I. El-Ghazawi, Tarek.

QA76.73. U63U63 2005
005.13′3 - - dc22 2004023262

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

About UPC

Many have contributed to the ideas and concepts behind the UPC language. The initial UPC language concepts and specifications were published as a technical report authored by William Carlson, Jesse Draper, David Culler, Katherine Yelick, Eugene Brooks, and Karen Warren in May 1999. The first UPC consortium meeting was held in Bowie, Maryland, in May 2000, during which the UPC language concepts and specifications were discussed and augmented extensively. The UPC consortium is composed of a group of academic institutions, vendors, and government laboratories and has been holding regular meetings since May 1999 to continue to develop the UPC language. The first formal specifications of UPC, known as v1.0, was authored by Tarek El-Ghazawi, William Carlson, and Jesse Draper and released in February 2001. The current version, v1.1.1, was released in October 2003 with minor changes and edits from v1.0. At present, v1.2 of the specifications is in the works and is expected to be released soon. v1.2 will be a publication of the UPC consortium because of the extensive contributions of many of the consortium members. v1.2 will incorporate UPC v1.1.1 with additions and will include the full UPC collective operations specifications, v1.0, and the I/O specifications v1.0. The first version of the UPC collective operations specification was authored by Steven Seidel, David Greenberg, and Elizabeth Wiebel and released in December 2003. The first version of the I/O specification was authored by Tarek El-Ghazawi, Francois Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross, and Dan Bonachea. It was released in July 2004. More information about UPC and the UPC consortium can be found at `http://upc.gwu.edu/`.

About This Book

Although the UPC specifications are the ultimate reference of the UPC language, the specifications are not necessarily easy to read for many programmers and do not include enough usage examples and explanations, which are essential for most readers. This book is the first to provide an in-depth interpretation of the UPC language specifications, enhanced with extensive usage examples and illustrations as well as insights into how to write efficient UPC applications.

The book is organized into eight chapters and five appendixes:

- Chapter 1 provides a quick tutorial that walks readers quickly through the major features of the UPC language, allowing them to write their first simple UPC programs.

- Chapter 2 positions UPC within the general domain of parallel programming paradigms. It then presents the UPC programming model and describes how data are declared and used in UPC.
- Chapter 3 covers the rich concept of pointers in UPC, identifying the types, declarations, and usage of the various UPC pointers and how they work with arrays.
- Chapter 4 explains how data and work can be distributed in UPC such that data locality is exploited through efficient data declarations and work-sharing constructs.
- Chapter 5 provides extensive treatment to dynamic memory allocation in the shared space, showing all options and their usages via many thorough examples.
- Chapter 6 covers thread and data synchronization, explaining the effective mechanisms provided by UPC for mutual exclusion, barriers, and memory consistency control.
- Chapter 7 provides sophisticated programmers with the tools necessary to write efficient applications. Many hand-tuning schemes are discussed along with examples and full case studies.
- Chapter 8 introduces the two UPC standard libraries: the collective operations library and the parallel I/O library.
- Appendix A includes the full UPC v1.1.1 specification.
- Appendix B includes the full UPC v1.0 collective library specifications.
- Appendix C has the full v1.0 UPC-IO specifications.
- Appendix D includes information on how to compile and run UPC programs.
- Appendix E is a quick UPC reference card that will be handy for UPC programmers.

Resources

The ultimate UPC resource is the consortium Web site, which is currently hosted at `http://upc.gwu.edu/`. For this book, however, the reader should also consult the publisher's ftp site, `ftp://ftp.wiley.com/public/sci_tech_med/upc/`, for errata and an electronic copy of the full code and Makefiles for all the examples given in the book. Additional materials for instructors wishing to use this book in the classroom are available from the first author.

Acknowledgments

Many of our colleagues have been very supportive during the development of this book. In particular, the authors are indebted to François Cantonnet, whose help has contributed significantly to the book's quality. The continuous cooperation and support of our editor, Val Moliere, and Dr. Hoda El-Sayed is also greatly appreciated.

CHAPTER 1

Introductory Tutorial

The objective of this chapter is to give programmers a general understanding of UPC and to enable them to write and run simple UPC programs quickly. The chapter is therefore a working overview of UPC. Subsequent chapters are devoted to gaining more proficiency with UPC and resolving the more subtle semantic issues that arise in the programming of parallel computing systems using UPC. In this chapter we introduce the basic execution model in UPC, followed by some of the key UPC features, including:

- Threads
- Shared and private data
- Pointers
- Distribution of work across threads
- Synchronization of activities between threads

More in-depth treatment of these subjects is provided in the respective book chapters. In addition, in subsequent chapters we address advanced features and usage that may be needed for writing more complex programs. Nonetheless, this introduction provides a valuable starting point for first-time parallel programmers and a good overview for more experienced programmers of parallel machines. However, advanced UPC programmers may wish to skip this chapter and proceed to the following chapters, as all material in this introduction is included and elaborated upon in the remainder of the book. It should be noted that UPC is an extension of ISO C [ISO99], and familiarity with C is assumed.

1.1 GETTING STARTED

UPC, or Unified Parallel C [CAR99, ELG01, ELG03] is an explicit parallel language that provides the facilities for direct user specification of program parallelism and control of data distribution and access. The number of threads, or degree of parallelism, is fixed at either compiler or program startup time and does not change

UPC: Distributed Shared Memory Programming, by Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick

midexecution. Each of these threads is created at run time and executes the same UPC program, although threads may take different paths through the program text and may call different procedures during their execution. UPC provides many parallel constructs that facilitate the distribution and coordination of work among these threads such that the overall task may be executed much faster in parallel than it would be performed sequentially.

Because UPC is an extension of ISO C, any C program is also a UPC program, although it may behave differently when run in a parallel environment. Consider, for example, a C program to print "hello world."

Example 1.1: helloworld1.upc

```
#include <stdio.h>

main()
{
  printf("hello world\n");
}
```

The program file should be created with a file name that ends in "`.upc`," such as "`helloworld1.upc`" in Example 1.1. The commands to compile and run the program may be platform-specific, but a typical installation may have a compiler that is named upcc and that is invoked by the following example command:

```
upcc –o hello –THREADS 4 helloworld1.upc
```

Compilation will then produce an executable file called `hello`, which will always run with four threads. Many machines require that parallel jobs be submitted to a special job queue or at least run with a special command, for example:

```
upcrun hello
```

This command will then produce the output lines

```
hello world
hello world
hello world
hello world
```

Each of the output lines above was produced by one of the four identical threads, each running the same `main` function. In parallel computing, this mode of operation is known as the *single program, multiple data* (SPMD) model, where all threads execute the same program but may be processing different data. Under the SPMD execution model all threads run concurrently from the start to the end of program execution, although there is no guarantee that they execute statements at the same rate, and in this example we cannot tell which thread produced which line of output.

We can change the number of threads by recompiling with a different ''–THREADS'' flag or by compiling without the flag and specifying the number of threads in the `upcrun` command. We can also determine the total number of threads at run time using the UPC identifier **`THREADS`** and identify the thread responsible for each line of output by using another identifier, **`MYTHREAD`**. In UPC, threads are given unique **`MYTHREAD`** identifiers from 0 to **`THREADS`**-1. Using these special constants, we produce a modified version of the ''hello world'' program in which the output indicates the total number of threads as well as which thread generated each line of output. In real parallel applications, **`MYTHREAD`** and **`THREADS`** are used to divide work among threads and to determine the thread that will execute each portion of the work. Incorporating these additional constructs, a new version of the ''hello world'' program is created.

Example 1.2: helloworld2.upc

```
#include <upc.h>
#include <stdio.h>

main()
{
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

In addition to supplying the thread identifiers to the `printf` statement, the inclusion of upc.h is provided, containing all pertinent UPC definitions. If the new file is compiled in the same manner as before and `hello` is executed, the following output may result:

```
Thread 1 of 4: hello UPC world
Thread 3 of 4: hello UPC world
Thread 2 of 4: hello UPC world
Thread 0 of 4: hello UPC world
```

The output lines do not necessarily appear in thread number order but may appear in any order (even ''normal'' ascending order!).

1.2 PRIVATE AND SHARED DATA

UPC has two different types of variables, those that are private to a given thread and those that are shared. This distinction also carries over to more general data types, such as arrays or structures. *Shared variables* are useful for communicating information between threads, since more than one thread may read or write to them. *Private variables* can only be accessed by a single thread but typically have some performance advantages over shared variables.

TABLE 1.1 Celsius–Fahrenheit Temperature Conversion Table

Fahrenheit	Celsius
32	0
50	10
68	20
86	30
104	40
122	50
140	60
158	70
176	80
194	90
212	100
230	110

To demonstrate the use of shared and private variables, consider the problem of printing a conversion table that provides a set of celsius temperatures and their corresponding Fahrenheit values as shown in Table 1.1. For now we ignore the problem of printing the table heading and of ordering the table elements, and instead, write a program that simply prints a set of valid Celsius–Fahrenheit pairs. Let us first consider a program in which each thread computes one table entry. The following program would produce the 12-entry table above, or some reordering of it, when run with 12 threads.

Example 1.3: temperature1.upc

```
#include <stdio.h>
#include <upc.h>

main ()
{
  static shared int step=10;
  int fahrenheit, celsius;

  celsius= step*MYTHREAD;
  fahrenheit= celsius*(9.0/5.0) + 32;

  printf ("%d \t %d \n", fahrenheit, celsius);
}
```

By default, variables in UPC are private, so the declaration

```
int fahrenheit, celsius;
```

creates instances of both variables for each thread. Each instance of the variables is independent, so the respective instance variables of different threads may have different values. They may be assigned and accessed within their respective thread without affecting the variable instances of other threads. Thus, each thread can be engaged in a separate computation without value conflicts while all threads are executing in parallel.

In contrast, the declaration

```
static shared int step=10;
```

creates a shared variable of type `int` using the UPC `shared` type qualifier. This means that there will be only one instance of `step`, and that variable instance will be visible and accessible by all threads. In the example, this is a convenient way to share what is essentially a constant, although UPC permits threads to write to shared variables as well.

Note that in the declaration, the type qualifier is `static`, as shared variables cannot have automatic storage duration. This ensures that shared variables are accessible throughout the program execution so that they cannot disappear when one thread exits a scope in which a shared variable was declared. Alternatively, the shared variable could have been declared as a global variable before `main()`.

The line

```
celsius = step * MYTHREAD;
```

accesses the `step` value to ensure that all threads will use celsius values that are multiples of 10, and use of `MYTHREAD` ensures that they will start at zero and be unique. The statements

```
fahrenheit = celsius * (9.0/5.0) + 32;
printf("%d \t %d \n", fahrenheit, celsius);
```

will be executed by each thread using local `celsius` and `fahrenheit` values. There is no guarantee for the order in which the threads will execute the print statement, so the table may be printed out of order. Indeed, one thread may execute all three of its statements before another thread has executed any. To control the relative ordering of execution among threads, the programmer must manage synchronization explicitly through the inclusion of synchronization constructs within the program code specification, which is covered in Section 1.4.

Example 1.3 is somewhat simplistic as a parallel program, since very little work is performed by each thread, and some of that work involves output to the screen, which will be serialized in any case. There is some overhead associated with the management of threads and their activities, so having just one computation per thread as in Example 1.2 is not efficient. Having larger computations at each thread will help amortize parallel overhead and increase efficiency. This small example and many others throughout the book are discussed because of their educational value and are not necessarily designed for high performance. The following example, however, allocates slightly more work to each thread.

Example 1.4: temperature2.upc

```
#include <stdio.h>
#include <upc.h>
#define TBL_SZ 12

main ()
{
  static shared int step=10;
  int fahrenheit, celsius, i;

  for (i=0; i< TBL_SZ; i++)
    if (MYTHREAD == i%THREADS)
    {
      celsius = step* i;
      fahrenheit = celsius* (9.0/5.0) + 32;
      printf ("%d \t %d \n", fahrenheit, celsius);
    }
  }
```

In Example 1.4, the number of entries in the table is given by the constant `TBL_SZ`, which is set to 12. The loop will sequence through all `TBL_SZ` iterations, assigning an iteration to each thread in round-robin fashion. Thus, thread 0 will execute iterations 0, `THREADS`, 2*`THREADS`, and so on, while thread 1 will execute iterations 1, `THREADS`+1, 2*`THREADS`+1, and so on. If the table size were 1, only thread 0 would execute an iteration; the rest of the threads would not do any useful work, as they all fail the test in the `for` loop.

The loop as written is not efficient, since each thread evaluates the loop header 13 times, and this redundant loop overhead may have nearly the same temporal cost as that of the sequential program. One way to avoid this is by changing the `for` loop as follows:

```
for(i=MYTHREAD; i < TBL_SZ; i+=THREADS)
```

In this case, each thread evaluates the loop header at most `TBL_SZ/THREADS` +1 times. Note that the `celsius` calculation now uses the loop index `i` rather than **`MYTHREAD`**, so it correctly evaluates several table entries.

1.3 SHARED ARRAYS AND AFFINITY OF SHARED DATA

A problem with the program of Example 1.4 is that the table may be produced out of order. One possible solution is to store the conversion table in an array and then have one thread print it in order. The following code shows how this might be done, although there is a remaining bug that we will fix in Section 1.4.

Example 1.5: temperature3.upc

```
#include <stdio.h>
#include <upc.h>
#define TBL_SZ 12

main ()
{
  static shared int fahrenheit[TBL_SZ];
  static shared int step=10;
  int celsius, i;

  for(i=MYTHREAD; i < TBL_SZ; i+=THREADS)
  {
    celsius= step*i;
    fahrenheit[i]= celsius*(9.0/5.0) + 32;
  }
  if(MYTHREAD==0)
    for (i=0 ; i < TBL_SZ; i++)
    {
      celsius= step*i;
      printf ("%d \t %d \n", fahrenheit[i], celsius);
    }
}
```

The line

```
static shared int fahrenheit[TBL_SZ];
```

declares an array `fahrenheit` of size `TBL_SZ` of integers, which will be shared by all threads. Thus, any of the threads can directly access any of the elements of `fahrenheit`. However, UPC establishes a logical partitioning of the shared space so that each variable in shared space is defined to have *affinity* to exactly one thread. On some platforms it is significantly faster for a thread to access shared variables to which it has affinity than to access shared variables that have affinity to another thread. A shared array such as `fahrenheit` will be spread across the thread partitions in round-robin fashion such that `fahrenheit[0]` has affinity to thread 0, `fahrenheit[1]` has affinitiy to thread 1, and so on. After each thread gets an element, we wrap around, giving `fahrenheit[THREADS]` to thread 0, `fahrenheit[THREADS+1]` to thread 1, and so on. This round-robin distribution of shared array elements is the default in UPC, but programmers may also distribute shared arrays by blocks of elements. In later chapters we show how to declare blocked distributions, which has a performance advantage for some applications. In this temperature-conversion example, however, the default distribution of the elements matches the work distribution, as each thread will compute exactly the table elements that have affinity to it.

In general, to maximize performance, each thread should be primarily responsible for processing the data that has affinity to that thread. This exercises two important features of UPC: control over data layout, and control over work distribution, both of which are critical to performance. On a machine with physically distributed memory, the UPC run-time system will map each thread and the data that has affinity to it to the same processing node, thereby avoiding costly interprocessor communication when the data and computation are aligned.

Shared scalar variables, such as `step` in Example 1.5, also have a defined affinity, which is always to thread 0. So the use of `step` in the initialization of celsius is likely to be less expensive on thread 0 than on all the others. Although the thread 0 default is not always what the programmers want, the clearly defined cost model allows them to optimize a UPC program in a platform-independent manner. For example, a thread may copy a shared variable into its own private variable to avoid multiple costly accesses. The body of the `for` loop will compute the Fahrenheit temperatures and store them in the `fahrenheit` array for printing later. This will be done by the last loop in the program, which is executed only by thread 0.

The erroneous assumption here is that since this printing loop follows the one that computes temperatures into `fahrenheit`, the results of the table will be printed in order. In fact, this does print the table in order; however, many of the entries of the table may hold the wrong answer. This is because printing will start as soon as thread 0 gets to the final print loop, while some of the other threads may be left behind and still executing the loop that computes the temperatures. This synchronization problem is addressed in Section 1.4.

1.4 SYNCHRONIZATION AND MEMORY CONSISTENCY

To guarantee that all threads finished computing the temperature table in the `fahrenheit` array before thread 0 starts printing the array, barrier synchronization is used. UPC offers several different types of barrier synchronization, described in Chapter 6, but the simplest is the `upc_barrier` statement. This is demonstrated in the following program, which now prints the values correctly, in order.

Example 1.6: temperature4.upc

```
#include <upc.h>
#define TBL_SZ 12

main ()
{
  static shared int fahrenheit[TBL_SZ];
  static shared int step=10;
  int celsius, i;
```

```
  for(i=MYTHREAD; i < TBL_SZ; i+=THREADS)
  {
    celsius = step*i;
    fahrenheit[i] = celsius*(9.0/5.0) + 32;
  }

  upc_barrier;

  if(MYTHREAD==0)
    for (i=0 ; i < TBL_SZ ; i++)
    {
    celsius= step*i;
    printf ("%d \t %d \n", fahrenheit[i], celsius);
    }
}
```

A `upc_barrier` statement ensures that all threads must reach that point before any of them can proceed further. Thus, if a thread arrives at the `upc_barrier` while any one of the other threads is still lagging behind, that thread will get blocked. Once all threads have arrived at the barrier, they all proceed past it. Thus, in our example we will be guaranteed that all threads have finished their computations and that the table is now holding the correct values before thread 0 begins executing the printing loop.

Barrier synchronization is not the only useful form of synchronization. Since shared data may be changed by any thread, there could be times when a thread wants to make sure that it has exclusive access to a shared data object, for example, to insert an element into a shared linked list or to update multiple values consistently in a shared array. In these situations a programmer may associate a lock with the data structure and acquire the lock before making a set of modifications to the structure. Only one thread may hold a given lock at any time, and if a second thread attempts to acquire the lock, it will block until the first thread releases it. In this way, programmers may guarantee mutual exclusion of shared data usage, preventing erroneous behavior that can result from having one thread modify a data structure while other threads are trying to access it. UPC provides powerful lock constructs for managing such shared data, which are described in Chapter 6.

In general, the classes of errors that arise in parallel programs from insufficient synchronization, called *race conditions*, occur when two threads access the same shared data at the same time and at least one of them modifies the data. Most programmers will be satisfied to write programs that are carefully synchronized using UPC locks and barriers to avoid race conditions. However, synchronization comes with a cost, and some programmers may wish to implement their own synchronization primitives from basic memory operations or write programs that read and write shared variables without synchronizing. These programmers are relying on the memory consistency model in the language, which ensures some basic properties of the memory operations. For example, if one thread writes a

shared variable while another reads it, the reading thread must see either the old or the new value, not a mixture of the two numbers, and if it keeps reading that variable, it will eventually see the new value. In general, the memory consistency model tells programmers whether operations performed by one thread have to be observed in order by other threads.

Memory performance is a critical part of overall application performance, and the memory consistency model can have a significant impact on that performance. For example, the memory consistency model affects the ability of the compiler to rearrange code and of the hardware to use caching and to pipeline and prefetch memory operations. UPC therefore takes the view that the programmer needs control over the memory consistency model and provides a novel set of mechanisms for this control, which are described in detail in Chapter 6.

1.5 WORK SHARING

Distributing work, typically independent iterations of a loop that can be run in parallel, is often referred to as *work sharing*. Although the use of **THREADS** and **MYTHREAD** in previous examples allowed us to distribute independent work across the threads, each computing a number of entries in the `fahrenheit` table, UPC provides a much more convenient iteration construct to do work sharing. This construct is called `upc_forall`. Example 1.7 can take advantage of this construct as shown below.

Example 1.7: temperature5.upc

```
#include <upc.h>
#define TBL_SZ 12

main ()
{
  static shared int fahrenheit[TBL_SZ];
  static shared int step=10;
  int celsius, i;

  upc_forall(i=0; i <TBL_SZ; i++; i)
  {
    celsius= step*i;
    fahrenheit[ i] = celsius* (9.0/5.0) + 32;
  }

  upc_barrier;
  if(MYTHREAD==0)
    for (i=0; i < TBL_SZ; i++)
    {
```

```
      celsius= step*i;
      printf ("%d \t %d \n", fahrenheit[i], celsius);
    }
}
```

In the line

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

the `upc_forall` construct has some syntactic similarities to the C language `for` loop, as the first three fields in `upc_forall` are almost identical to the corresponding fields of the C `for` loop. The first field, `i=0`, initializes the counter variable, i. The second field, `i<TBL_SZ`, provides the test that determines if the variable value is within the range specified. The third field, `i++`, specified the increment value separating successive values of the counter variable, thus determining how it is updated. So these fields simply identify the first iteration, test whether the last iteration is reached, and increment the iteration counter. The `upc_forall` construct differs from its sequential counterpart by a fourth field, which is called *affinity*. In this case, the affinity `i` indicates that iteration `i` will be performed by thread (`i` modulo **`THREADS`**). Thus, iteration distribution across the threads will take place in round-robin fashion, in just the same way that the array elements themselves were distributed by default. As the iteration number and the array index are the same, each thread will be processing only the array elements that have affinity to it. The performance implication is that threads will probably find the data they will be processing locally accessible and will therefore avoid costly remote access and the substantial overhead that this may require.

Note that after the `upc_forall` statement, we still used a barrier synchronization. This is because the UPC specification does not require an implicit barrier at the end of the iteration statement. The `upc_forall` has interesting and powerful additional options and can be used in many different ways, providing significant flexibility of control, as discussed later in the book.

1.6 UPC POINTERS

Pointers have been one of the most interesting and useful concepts of the C programming language. It is perhaps difficult to imagine a C application program, even a parallel one, without pointers. For now, let us consider replacing the array notation in Example 1.7 with its equivalent pointer representation. As a first step, let us do that in the printing loop only.

Example 1.8: temperature6.upc

```
#include <upc.h>
#define TBL_SZ 12
```

```
main ()
{
  static shared int fahrenheit[TBL_SZ] ;
  shared int *fahrenheit_ptr=fahrenheit;
  static shared int step=10;
  int celsius, i;

  upc_forall(i=0; i <TBL_SZ; i++; i)
{
    celsius= step*i;
    fahrenheit[i] = celsius*(9.0/5.0) + 32;
  }

  upc_barrier;

  if(MYTHREAD==0)
    for (i=0 ; i < TBL_SZ ; i++)
    {
      celsius= step*i;
      printf ("%d \t %d \n", *fahrenheit_ptr++, celsius);
    }
}
```

The line

```
shared int *fahrenheit_ptr=fahrenheit;
```

declares `fahrenheit_ptr` to be a pointer to type shared `int` and initializes that pointer to point at the first element of the shared array `fahrenheit`. The pointer `fahrenheit_ptr` is actually a private pointer to a shared type. This means that each thread will have an independent copy of the pointer `fahrenheit_ptr`, which is able independently to advance and access the elements of `fahrenheit`. Initially, all these copies of `fahrenheit_ptr`, one per thread, will be pointing at the first element of `fahrenheit`.

The line

```
printf ("%d \t %d\n", *fahrenheit_ptr++, celsius);
```

de-references the pointer printing the corresponding contents and then advances the `for` loop pointer variable to designate the next element in the array. This, will be executed only by thread 0, according to the construct

```
if(MYTHREAD==0)
```

In the following example, we extend our use of pointers to replace all array notations with pointer notations and make needed adjustments to the code.

Example 1.9: temperature7.upc

```
#include <upc.h>
#define TBL_SZ 12

main ()
{
  static shared int fahrenheit[TBL_SZ] ;
  shared int *fahrenheit_ptr;
  static shared int step=10;
  int celsius, i;

  fahrenheit_ptr = fahrenheit + MYTHREAD;

  upc_forall(i=0; i <TBL_SZ; i++; i)
  {
    celsius = step* i;
    *fahrenheit_ptr = celsius*(9.0/5.0) + 32;
    fahrenheit_ptr += THREADS;
  }

  upc_barrier;

  if(MYTHREAD==0)
  {
    fahrenheit_ptr=fahrenheit;
    for (i=0; i < TBL_SZ ; i++, fahrenheit_ptr++)
    {
      celsius= step* i;
      printf ("%d \t %d \n", * fahrenheit_ptr, celsius);
    }
  }
}
```

The line

```
shared int * fahrenheit_ptr;
```

declares `fahrenheit_ptr` to be a pointer to a shared variable. However, `fahrenheit_ptr` itself is private and each thread has an independent instance of it. In the lines

```
fahrenheit_ptr = fahrenheit + MYTHREAD;
```

and

```
fahrenheit_ptr += THREADS;
```

each of the `fahrenheit_ptr` instances is initialized to point at the first array element that has the same affinity as the pointer instance itself. In addition, the update advances each pointer by **`THREADS`** elements in each iteration, to move to the next element that has affinity to the thread of that pointer instance.

UPC has other types of pointers. For example, private pointers to private data follow from being an ISO C compliant and a superset. The language also allows the use of shared pointers to shared data. Casting from one type of pointer to another is possible. All these issues are handled in more detail in Chapter 3.

1.7 SUMMARY

In this chapter we introduced the basic concepts of UPC in a tutorial style to enable programmers to write their first UPC code quickly. We have in particular demonstrated that UPC is a superset of C, and all C programs will run under UPC. However, this will naturally create several copies of the same program running in the SPMD mode.

Under UPC, multiple threads will be operating independently and each thread may have access to both private and shared data objects, variables, and arrays. A private variable has one independent instance per thread. The total number of threads is **`THREADS`**, and each thread identifies itself using **`MYTHREAD`**. **`THREADS`** and **`MYTHREAD`** can be thought of as special constants. Shared scalars have affinity with thread 0. Shared array elements, however, are distributed by default in round-robin fashion across the threads.

UPC has many synchronization constructs for barrier, split-phase barrier, locks, and fence. UPC also provides programmers with the ability to specify the memory consistency model as relaxed or strict. Work can be distributed based on **`THREADS`** and **`MYTHREAD`**. Work can be distributed conveniently, however, using `upc_forall`. All iterations must be independent in order to use `upc_forall`. UPC provides rich pointer concepts. Threads can point to shared data using either shared or private pointers. In addition, C pointer declarations result in private pointers to private data. It is possible under UPC to cast one type of pointer to another.

EXERCISES

1.1 Create a sequential C version of the temperature table generation program, to compute Fahrenheit temperatures from 0 to 1000 degrees Celsius by steps of 0.01 degree. Comment the `printf` line and use appropriate system calls to measure the wall clock time for program execution by measuring the times at the beginning and end of the program. Compile and run using cc for an adequately large table that gives some measurable execution time, and note the execution time. Compile using upcc with one thread and run. Compare and comment on the measured time for the sequential program when compiled by cc versus upcc.

1.2 Create a UPC parallel program for generating the temperatures table by using the improved for loop into the last parallel example given in Section 1.4. Comment the `printf` and add the time measurement statements as in Exercise 1.1. Compile using upcc and run with one thread. Compare the results of running the UPC program with one thread to those of Exercise 1.1.

1.3 Rewrite the program of Exercise 1.2 using a two-dimensional shared array of two rows, where the first row holds the Celsius temperatures and the second row holds the corresponding Fahrenheit temperatures.

1.4 Write an UPC program to sum the elements of two shared vectors. Make sure that each thread operates only on the array elements that have affinity to that thread.

1.5 Write a program that computes the mean of all elements in a shared array of a general size, larger or smaller than the number of threads. You can use another shared array of size equal to the number of threads to hold the partial sums from each thread. At the end, thread 0 will need to sum up all partial sums, compute the mean, and print the result.

1.6 Repeat Exercise 1.4 using the `upc_forall` construct.

1.7 Repeat Exercise 1.5 using the `upc_forall` construct.

CHAPTER 2

Programming View and UPC Data Types

Parallel programming languages that are available today represent a diversity of programming models. Depending on the physical structure and incorporated mechanisms of the underlying parallel computer, one or more languages may be preferable to others in both ease of programming and/or delivered performance. Similarly, the organization of the data structures and the flow control of the tasks of a given application algorithm may strongly influence the parallel programming language to be employed. UPC is one such parallel programming language that facilitates general-purpose parallel computing through a set of constructs particularly well suited to the major classes of parallel computers and a wide range of parallel applications. In this chapter we present the foundation principles of parallel programming as reflected by some of the most widely used languages and introduce UPC from the perspective of these same basic concepts to position UPC in the domain of parallel programming. Details of the UPC programming model are presented with a discussion of the memory sharing and thread execution view. The remainder of this chapter covers basic declarations, types, associated storage, and constraints in the light of the UPC memory sharing and execution model.

2.1 PROGRAMMING MODELS

A programming model is simply the abstract view of how data and instructions are stored and how processing takes place as perceived by the programmer [HWA98]. In uniprocessor systems, it is fair to say that there is one basic programming model, which is the von Neumann stored program model. Under this model, there is only one memory, and all data and instructions are stored in it. The processor fetches and decodes the program instructions and accesses and processes data accordingly. In a parallel system, the architecture is more complex, due to the multiplicity of processors and possibly, memory subsystems. Parallel programming models therefore

UPC: Distributed Shared Memory Programming, by Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick