UPC
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 Architecture / 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
4.4 Distributing Trees 62
4.5 Summary 71
Exercises 71

5. Dynamic Shared Memory Allocation 73

5.1 Allocating a Global Shared Memory Space Collectively 73
5.2 Allocating Multiple Global Spaces 78
5.3 Allocating Local Shared Spaces 82
5.4 Freeing Allocated Spaces 89
5.5 Summary 90
Exercises 90

6. Synchronization and Memory Consistency 91

6.1 Barriers 92
6.2 Split-Phase Barriers 94
6.3 Locks 99
6.4 Memory Consistency 108
6.5 Summary 113
Exercises 114

7. Performance Tuning and Optimization 115

7.1 Parallel System Architectures 116
7.2 Performance Issues in Parallel Programming 120
7.3 Role of Compilers and Run-Time Systems 122
7.4 UPC Hand Optimization 123
7.5 Case Studies 128
7.6 Summary 135
Exercises 135

8. UPC Libraries 137

8.1 UPC Collective Library 137
8.2 UPC-IO Library 141
8.3 Summary 146

References 147

Appendix A: UPC Language Specifications, v1.1.1 149
Appendix B: UPC Collective Operations Specifications, v1.0 183
Appendix C: UPC-IO Specifications, v1.0 203
Appendix D: How to Compile and Run UPC Programs 243
Appendix E: Quick UPC Reference 245
Index 251
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
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

```c
#include <stdio.h>

int 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.
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

```c
#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

```c
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

```c
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

```c
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

```c
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 syn-
chronization 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.