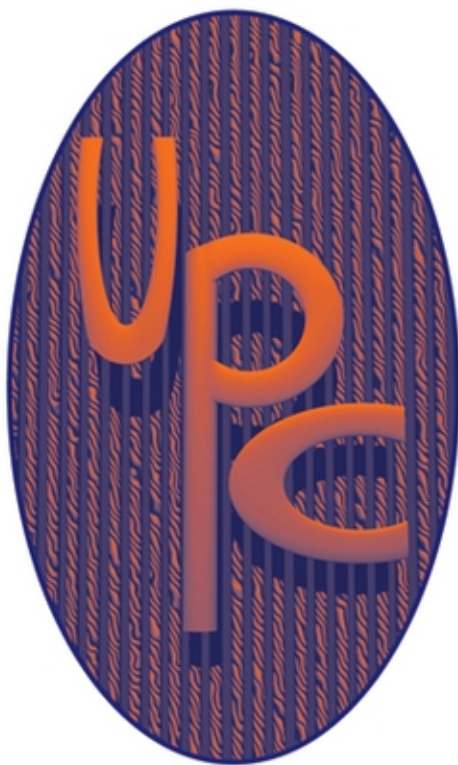


Wiley Series on Parallel and Distributed Computing  
Albert Y. Zomaya, Series Editor

 WILEY

# 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



**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 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



A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2005 by John Wiley & Sons, Inc. All rights reserved.

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

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400, fax 978-646-8600, or on the web at [www.copyright.com](http://www.copyright.com). Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

**Limit of Liability/Disclaimer of Warranty:** While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic format.

***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

---

<b>Preface</b>	<b>vii</b>
<b>1. Introductory Tutorial</b>	<b>1</b>
1.1 Getting Started	1
1.2 Private and Shared Data	3
1.3 Shared Arrays and Affinity of Shared Data	6
1.4 Synchronization and Memory Consistency	8
1.5 Work Sharing	10
1.6 UPC Pointers	11
1.7 Summary	14
Exercises	14
<b>2. Programming View and UPC Data Types</b>	<b>17</b>
2.1 Programming Models	17
2.2 UPC Programming Model	20
2.3 Shared and Private Variables	21
2.4 Shared and Private Arrays	23
2.5 Blocked Shared Arrays	25
2.6 Compiling Environments and Shared Arrays	30
2.7 Summary	30
Exercises	31
<b>3. Pointers and Arrays</b>	<b>33</b>
3.1 UPC Pointers	33
3.2 Pointer Arithmetic	35
3.3 Pointer Casting and Usage Practices	38
3.4 Pointer Information and Manipulation Functions	40
3.5 More Pointer Examples	43
3.6 Summary	47
Exercises	47
<b>4. Work Sharing and Domain Decomposition</b>	<b>49</b>
4.1 Basic Work Distribution	50
4.2 Parallel Iterations	51
4.3 Multidimensional Data	54



4.4	Distributing Trees	62
4.5	Summary	71
	Exercises	71
<b>5.</b>	<b>Dynamic Shared Memory Allocation</b>	<b>73</b>
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
<b>6.</b>	<b>Synchronization and Memory Consistency</b>	<b>91</b>
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
<b>7.</b>	<b>Performance Tuning and Optimization</b>	<b>115</b>
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
<b>8.</b>	<b>UPC Libraries</b>	<b>137</b>
8.1	UPC Collective Library	137
8.2	UPC-IO Library	141
8.3	Summary	146
	<b>References</b>	<b>147</b>
	<b>Appendix A: UPC Language Specifications, v1.1.1</b>	<b>149</b>
	<b>Appendix B: UPC Collective Operations Specifications, v1.0</b>	<b>183</b>
	<b>Appendix C: UPC-IO Specifications, v1.0</b>	<b>203</b>
	<b>Appendix D: How to Compile and Run UPC Programs</b>	<b>243</b>
	<b>Appendix E: Quick UPC Reference</b>	<b>245</b>
	<b>Index</b>	<b>251</b>

### 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/](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.

# 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

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 affinity 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 inter-processor 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.



# 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

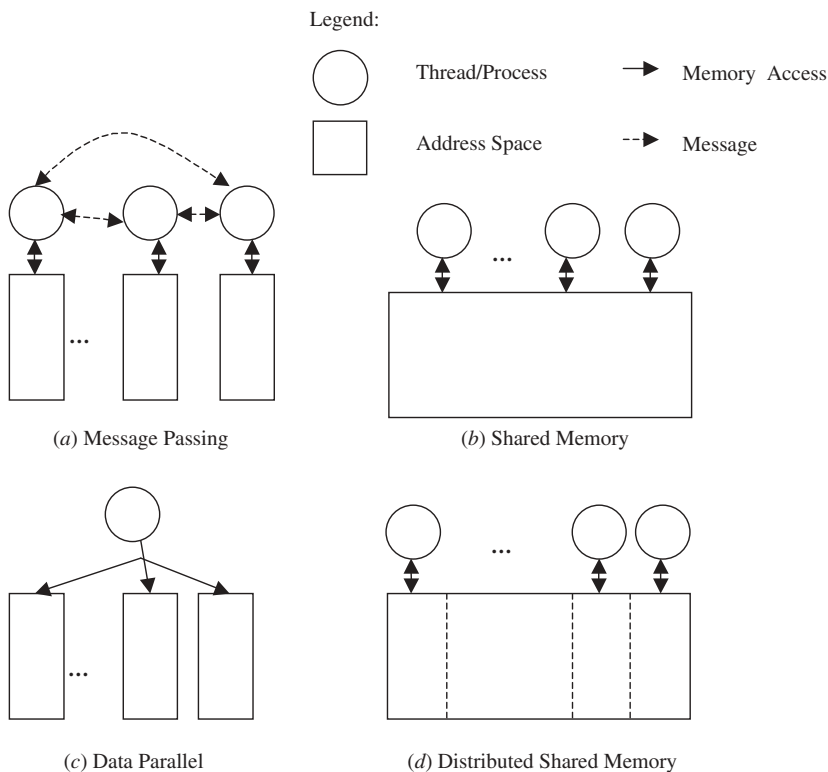
A programming model is simply the abstract view of how data and instructions are stored and how processing takes place as perceived by the programmer [HWA98]. In uniprocessor systems, it is fair to say that there is one basic programming model, which is the von Neumann stored program model. Under this 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

also enable the programmer to express how the application should be decomposed (i.e., how data and work will be distributed for parallel execution).

Parallel programming models expose common architecture features to enable efficient mapping of the programs onto the architectures. However, they should be independent of precise details of specific parallel architectures, to allow mapping of any parallel programming model onto a variety of parallel architectures. Programming models should, however, remain simple for ease of use.

Popular parallel programming models include message passing, shared memory, data parallel, and the distributed shared memory model. UPC uses the distributed shared memory programming model. In this section we distinguish among these models and give examples of their implementations. Our intent is to position UPC in the world of parallel programming, highlighting powerful features.

In the *message-passing model* (Figure 2.1a), parallel processing is derived from the use of a set of concurrent sequential processes cooperating on the same task. As each process has its own private space, two-sided communication in the form of sends and receives is used. This results in substantial overhead in interprocessor communications, especially in the case of small messages. With separate spaces, ease of use becomes another concern. As large problems are decomposed for parallel



**Figure 2.1** Address Space and Execution in Parallel Programming Models

processing, the overall view of the problem is lost and replaced by multiple private ones, placing a bigger burden on the program to maintain such global nature of the application, which may require adding more code. The most popular example of message passing is MPI, the message-passing interface [MPI94, SNI98].

Another popular programming model is the *shared memory model* (Figure 2.1b). The view provided by this model is one in which multiple independent threads operate in a shared space. The most popular implementations of this model are OpenMP [OPE97] and Pthreads [BUT97]. This model is characterized by its ease of use, as programmers need not treat remote memory accesses differently from local accesses. An expression, for example, can imply a remote memory read if any of its variables is located on a remote computing element or memory bank of the physically distributed system. Similarly, an assignment statement can cause a remote memory write. Since all concurrent threads see a single shared memory space, the application view remains integrated and similar to that of the sequential case. A negative consequence of this is that due to threads being unaware of whether the data they are processing is local or remote, unnecessary remote memory accesses might be generated, resulting in performance degradation. Therefore, under the pure shared memory programming model, it is difficult to exploit the inherent data locality in applications to achieve the highest efficiency.

Another programming model, the *data parallel model* shown in Figure 2.1c, derives its concurrency from processing many data items simultaneously in the same manner. This model employs only one executing process, for which every operation executed processes multiple data items identically. The major problem with this model is that it does not permit independent branching within the executing process to allow processing different data in different ways. Thus, applications that are richer in functional parallelism than in data parallelism may not be expressed effectively under this model. Examples of languages that followed such a scheme are C\* [ROS87] and HPF [HPF97].

The last model that we discuss here is the *distributed shared memory (DSM) programming model* (Figure 2.1d), also called the *partitioned global address space (PGAS) model*, which has been adopted by UPC. This model can achieve the desired balance between ease of use and exploiting data locality while avoiding the problem of independent branching in the data parallel model. Under this model, independent threads are operating in a shared space, just as in the shared memory model. However, this shared space is logically partitioned among the threads. This enables mapping to the same physical node of each thread and the data space that is associated with it locally. Programmers can thus declare the data to be processed by a given thread in the space partition that has affinity to that thread. Exploiting locality of access in this manner eliminates or minimizes unnecessary remote accesses from the beginning. Further, the multiple threads of the DSM programming model can all be executing the same program in *single program, multiple data stream (SPMD)* fashion that relaxes the rigid flow control of the data parallel programming model and thus avoids the independent branching problem. Thus, the DSM model provides a good balance between program abstraction for ease of use and portability, on the one hand, and direct control of resource management

for good performance, on the other hand, to achieve efficient execution with contemporary computer architectures.

## 2.2 UPC PROGRAMMING MODEL

Having defined the space of programming models, UPC can be positioned in this conceptual domain by defining the UPC execution and data sharing environment, which comprises the principal semantic elements of the UPC programming model. Figure 2.2 illustrates the memory and execution model as viewed by UPC codes and programmers. Under UPC, a number of threads work independently, with no implicit synchronization, except that all threads start and finish together. This implies barrier synchronization at the beginning and end of a program. The total number of threads is given by the integer **THREADS**, and each thread can identify itself using **MYTHREAD**. Thus, **MYTHREAD** and **THREADS** can be thought of as a private constant at each thread and a global constant visible to all threads, respectively. The total number of threads, **THREADS**, can be specified at either compile time or run time, using the compile or the run command line, respectively.

UPC works in SPMD style, where each thread executes the same `main()` function. This does not limit the flexibility of the execution model, because conditional flow control within the thread can direct different thread instances to perform different parts of the total thread code based on the **MYTHREAD** identifier and intermediate result data values.

UPC represents a variant instance of the DSM paradigm with additional private address spaces for local computations. Under UPC, memory is composed of a logically partitioned shared memory space and additional private memory spaces, as shown in Figure 2.2. All distributed threads can reference any address location (shared variables) in the shared data space, but only its own private data space. The shared space, however, is logically divided into portions, each with a special association (affinity) to a given thread. In this way, UPC enables the programmer, with proper declarations, to keep the shared data that will be dominantly processed by a given thread (and occasionally, accessed by others) associated with that thread.

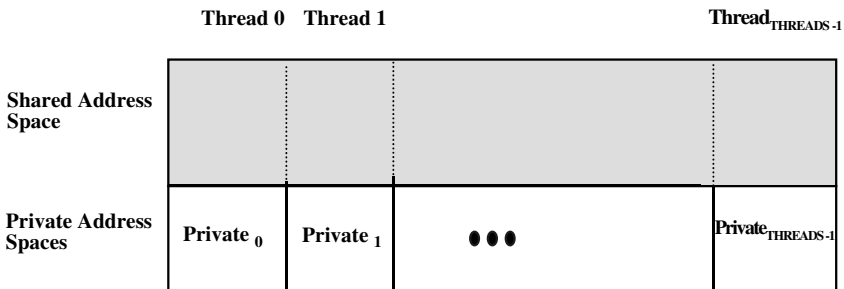


Figure 2.2 UPC Memory and Execution Model

Thus, a thread and the data that has affinity to it can be mapped by the system into the same physical node. This provides programmers with the necessary language constructs to exploit inherent data locality in their applications. Although an implementation may map each UPC thread to a different CPU, the UPC specification does not prohibit mapping more than one thread to the same CPU, permitting the exploitation of multithreading hardware support (such as in Tera MTA [ALV90]) when available.

UPC allows dynamic memory allocation in the shared space. Dynamic allocations in the private space are inherited from ISO C, as UPC is only an extension of ISO C. In addition, UPC has pointers that can access the shared address space. Pointers into the private spaces are also supported by UPC, as it is an ISO C extension. In conjunction with the programming model, UPC provides explicit extensions to the syntax and semantics of ISO C to facilitate expressing parallel applications. Therefore, UPC can be classified as an explicit parallel extension to ISO C that follows the distributed shared memory programming model.

### 2.3 SHARED AND PRIVATE VARIABLES

As discussed earlier, UPC is an extension of ISO C. Its execution follows an SPMD model in which each thread executes `main()`. A given thread in UPC appears as a sequential ISO C program. In UPC, an object can be declared as shared or private. A private object would have a separate instance for each thread, equivalent in structure and local identifiers but different in value among the distinct threads. One particular thread, thread 0, is distinguished among all others. It is allocated all scalar shared objects such that their affinity is assigned to thread 0. The following two examples show scalar private and shared declarations.

By default, any ISO C style declaration under UPC results in private objects. For example, the declaration

```
int x;    // x is private, one x in the private space of each thread
```

creates one private variable `x` for each thread. Each thread can only reference and manipulate its own instance of `x`. This is consistent with the fact that UPC is simply an extension of ISO C in which parallel execution follows the SPMD model, with each thread executing `main()`. This demonstrates how UPC flows logically from ISO C augmented with a number of basic parallel programming constructs. These explicit extensions to the syntax and semantics of C provide the user with the means to specify and control parallel execution from within the parallel application program.

Declaring an object to be shared, however, requires explicit use of the **shared** qualifier. For example:

```
shared int y;    // y is shared, only one y at thread 0 in the shared space
```



creates one instance of the variable `y`, which can be referenced and manipulated by all threads. The variable `y` is therefore created in the shared space and has affinity to thread 0, as it is a scalar variable. Scalar shared objects in UPC will always have affinity to thread 0. This is logically consistent with the fact that the first element of a shared array always has affinity to thread 0, as explained in the next section.

Shared variables cannot have automatic storage duration: as, for example, in

```
void foo(void)
{
    shared int x;           // not allowed
    static shared int y;   // allowed
    shared int *p;         // allowed
    int *shared q;         // not allowed
    ...
}
```

In the previous `foo()` function, the first declaration,

```
shared int x;
```

is not allowed since it has an automatic storage duration, as it appears inside the function. However, `static shared` variables are allowed, with the example of `y`.

Although pointers are treated thoroughly in Chapter 3, the third and last declarations involve pointer declarations that bring up useful points to the discussion. In the third declaration,

```
shared int *p;
```

is a private pointer to shared, which gives each thread an independent pointer, stored in its private space but pointing into the shared space. Thus, **shared** in this context qualifies the pointed to variable, and it creates only private pointers to it. This declaration is therefore allowed since the pointers themselves are not shared.

In the last declaration,

```
int *shared q;
```

is a shared pointer to private. Thus, **shared** in this context qualifies the pointer that is to be created in the shared space and therefore is not allowed. It should be noted, however, that although the previous declaration helps explain an important concept, use of shared pointers to private variables is strongly discouraged. In general, **shared** should not appear in a declarator that has automatic storage duration, except when it results in creating storage in the private space.

One remedy is to move the first and last declarations outside the function `foo()` body, as follows:

```

shared int x;
int *shared q;

void foo(void)
{
    shared int *p;
    ...
}

```

Similarly, no data can be declared as shared inside a private structure except for private pointers to shared. Shared structures, on the other hand, can only contain data that are shared. For example:

```

struct rectangle {
    shared int width;
    shared int length;
} myrect;

```

is illegal, as both `width` and `length` are shared-qualified, whereas the structure is not. However,

```

shared struct rectangle {
    int width;
    int length;
} myrect;

```

is allowed since the structure is shared-qualified, as is

```

struct foo {
    shared int *p;    // allowed
    int *shared q;   // not allowed
};

```

Type conversion among shared and private is possible. It can be accomplished through cast and assignments. In general, private objects cannot be cast to shared objects, and assignment of private to shared objects has undefined results. These type conversions can be quite useful in the case of private and shared pointers and will be addressed further in that context. Shared-qualified objects may also have a reference qualifier to define their behavior from a memory consistency point of view. This is treated in Chapter 6.

## 2.4 SHARED AND PRIVATE ARRAYS

Shared arrays are created in the shared space. By default, elements of a shared array will be distributed across the threads in round-robin fashion. In other words, the first element is created in the shared space that has affinity to thread 0, the second

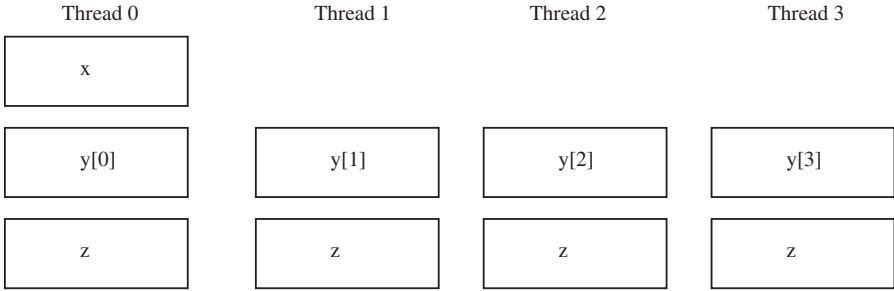


Figure 2.3

element in the shared space that has affinity to thread 1, and so on. For simplicity, however, we say that the first element goes to thread 0, the second to thread 1, and so on. The following example declarations demonstrate how a shared vector declaration behaves compared to shared scalar and private scalar declarations.

The declarations

```
shared int x; /* x is a shared scalar and
              will have affinity to thread 0 */
shared int y[THREADS]; /* shared array */
int z; /* private scalar */
```

in the case of four threads will result in the default layout shown in Figure 2.3, where *x* and *y* were created in the shared space, and instances of *z* were created in the private spaces of each thread. Thus, if the declaration

```
shared int y[THREADS];
```

was replaced with

```
int y[THREADS];
```

each thread would have had its own full four-element private version of the array *y*.

In the case of higher-dimensional arrays, the elements of a shared array are still distributed in round-robin fashion. Consider, for example, the declaration

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

This declaration results in the layout shown in Figure 2.4. Such default distribution can be very useful, as the vector addition program in the next example shows.

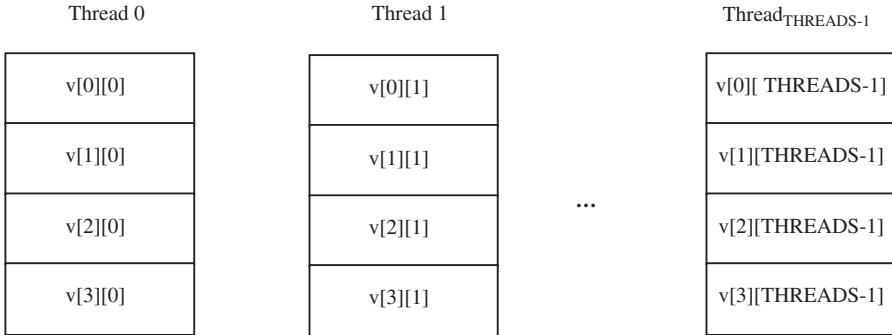


Figure 2.4

**Example 2.1:** vectadd1.upc

```
#include <upc_relaxed.h> //upc_relaxed explained in Chapter 6
#define N 10 * THREADS

shared int v1[N] , v2[N] , v1plusv2[N] ;

int main ()
{
    int i;

    upc_forall ( i = 0; i < N; i ++; i )
        v1plusv2 [i] = v1 [i] + v2 [i] ;

    return 0;
}
```

In this example, two vectors  $v1$  and  $v2$  are added to produce the vector  $v1plusv2$ . By default,  $v1$ ,  $v2$ , and  $v1plusv2$  are distributed across the threads in round-robin fashion. The last field in the `upc_forall` statement is the affinity field and says that iteration  $i$  should be executed on thread  $i\% \text{THREADS}$ . Thus, iterations are also distributed in round-robin fashion across the threads, which will ensure that each thread manipulates only the shared data that are available locally.

**2.5 BLOCKED SHARED ARRAYS**

In many cases, the default shared array distribution is not appropriate for optimal execution, and an alternative explicit data layout may improve data locality exploitation and execution efficiency. Consider the following matrix–vector multiplication example.

**Example 2.2:** matvect1.upc

```
#include <upc_relaxed.h>

shared int a [THREADS] [THREADS];
shared int b [THREADS] , c [THREADS];

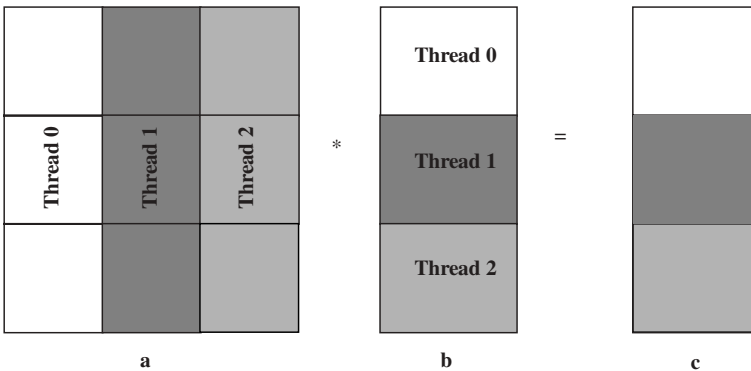
int main (void)
{
    int i, j;

    upc_forall(i = 0; i < THREADS; i++; i)
    {
        c [i] = 0;
        for (j= 0; j <THREADS; j++)
            c [i] += a [i] [j] *b [j] ;
    }
    return 0;
}
```

The default layout in the shared space in the case of running with three threads is shown in Figure 2.5. A first glance at the layout will show that for computing each  $c [ ]$  element, there will be more remote memory accesses than local accesses. For example, in case of  $c [0]$  , only  $a [0] [0]$  and  $b [0]$  are local input operands; the remaining four are not. Luckily, this default distribution can be altered to better fit the need of an application and improve locality exploitation in the underlying computations.

The default shared array distribution can be altered by specifying a given block size, also called a *blocking factor*, in the declaration using a layout qualifier as follows:

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



**Figure 2.5**

For example:

```
shared[4] int a[16];
```

In the previous case, array `a [ ]` will have 16 elements distributed across the threads in four-element by four-element blocks in a round-robin fashion. Thus, the block size and total number of threads, **THREADS**, determine the affinity of each data element to threads as follows: Element `i` of a blocked array has an affinity to thread:

$$\left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \bmod \text{THREADS}$$

Thus, the declaration

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

has a layout qualifier of 3, which means that array entries will be blocked and distributed across the threads in blocks of three in round-robin fashion. The resulting layout for **THREADS** = 3 is shown in Figure 2.6. Thus, if the previous declaration were changed to

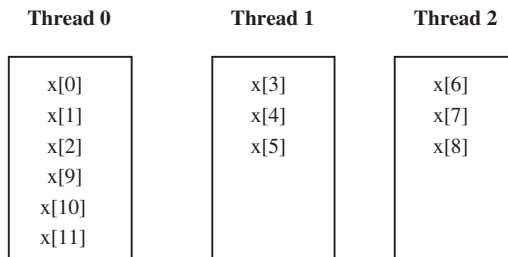
```
shared[12] int x[12];
```

all array elements would have affinity to thread 0. This can also be established using the *indefinite block size*. Omitting the block size or making it zero in the layout qualifier brackets would result in making all array elements have affinity to thread 0. Using such indefinite block size, the effect of the previous declaration can be established through the declaration

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

or

```
shared[0] int x[12];
```



**Figure 2.6**

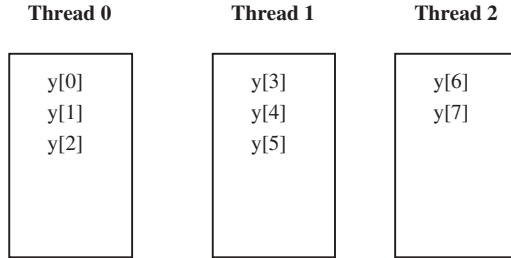


Figure 2.7

In many cases it is desirable to distribute an array of data in contiguous blocks such that, when possible each thread gets one of those chunks. One convenient way to do this is by using the \* layout qualifier. For example,

```
shared[*] int y[8];
```

would produce the layout shown in Figure 2.7 for the case of three threads.

Layout qualifiers work in the same way with two- and higher-dimensional arrays as in the case of one-dimensional arrays. Consider, for example, the declaration

```
shared[3] int A[4][4];
```

In this case, array elements are also blocked by a factor of 3. Therefore, blocks of three elements each are distributed across the threads in round-robin fashion until all the array elements are allocated. The resulting layout in the case of four threads is shown in Figure 2.8.

By employing the foregoing techniques, the matrix–vector multiplication example can now be revisited.

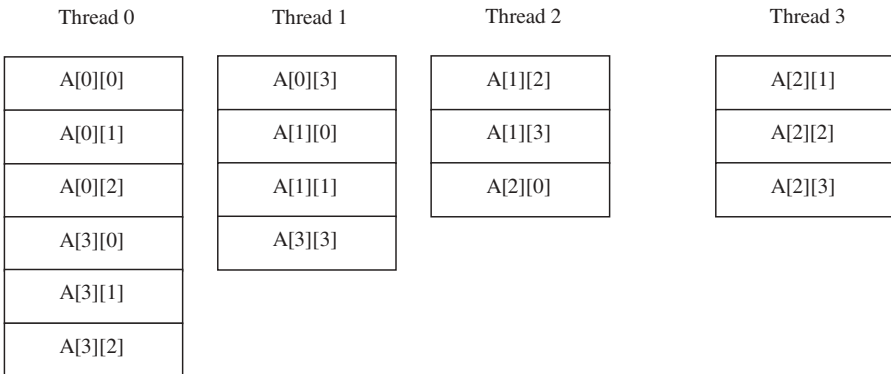


Figure 2.8

**Example 2.3:** matvect2.upc

```

#include <upc_relaxed.h>

shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

int main (void)
{
    int i, j;

    upc_forall (i = 0; i < THREADS; i++; i)
    {
        c[i] = 0;
        for (j= 0; j < THREADS; j++)
            c[i] += a[i][j] * b[j] ;
    }

    return 0;
}

```

The only difference between this version and the version that was shown in Example 2.2 is in the declaration of `a[ ]`, where `a[ ]` is now blocked by a factor equal to the number of **THREADS**. The resulting layout for the case of three threads is shown in Figure 2.9. In this case, computing `c[0]` requires only two remote accesses for reading `b[1]` and `b[2]` instead of the four remote accesses needed in the case of default distribution.

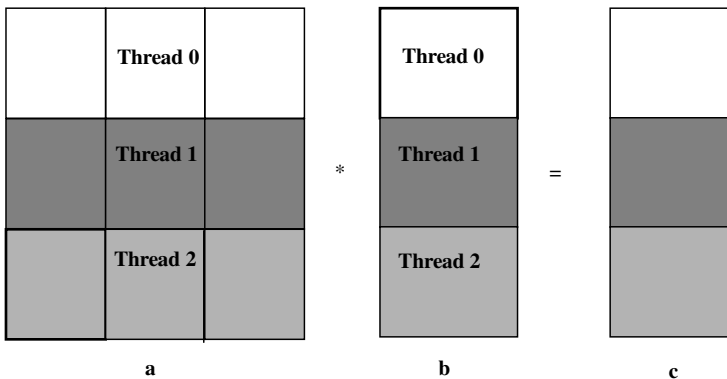


Figure 2.9



## 2.6 COMPILING ENVIRONMENTS AND SHARED ARRAYS

The number of UPC executing threads can be specified at either compile time or run time. If the number of threads is known at compile time, the UPC program is translated in the “static THREADS” environment. If the number of threads is not known at compile time, the UPC program is translated in the “dynamic THREADS” environment.

Specifying the number of threads at compile time gives compilers the opportunity to offer their best optimizations. On the other hand, it may not be very convenient when a program is to be run with different numbers of threads. Specifying the number of threads at run time allows the users to compile once and run multiple times with different numbers of threads. However, this can present technical challenges to compilers, particularly where the number of array elements allocated to each thread depends on the number of executing threads. To contain such challenges, a number of constraints are established. One clear constraint is that an array declaration is considered illegal if the UPC program is translated in the “dynamic THREADS” environment (i.e., **THREADS** is specified at run time, and the number of elements to allocate at each thread depends on the number of threads). For example, the declarations

```
shared int x [10*THREADS];
shared [ ] int x [10];
```

are legal in both the “static THREADS” and “dynamic THREADS” translation environments. In the first case, 10 elements will be allocated to each thread, and in the second declaration 10 elements will be allocated to thread 0.

On the other hand, the declarations

```
shared int x [10];
shared [ ] int x [THREADS];
shared int x [10+THREADS];
```

are illegal in the case of the “dynamic THREADS” translation environment. In the first declaration, it is unclear how many `x [ ]` elements will be given to each thread without knowing the total number of threads. In the second declaration, although the entire array goes to thread 0, the size of that array is unknown without knowing the total number of threads. In the third declaration, it is clear that each thread gets at least one element, but without knowing the total number of threads, the total share of each thread of the array elements will be unknown.

## 2.7 SUMMARY

UPC is an extension of ISO C that follows the distributed shared memory programming model. All threads can access the entire shared space. However,

the shared space is logically partitioned, and each partition has affinity to one of the threads. This enables programmers to co-locate data and processing, thus exploiting data locality for improved performance. Each thread has, in addition, a private memory space, which should be accessed by that thread only. UPC operates in a SPMD model of execution, where all threads execute the `main()` function. No synchronization is implied except that threads must start and finish the same program at the same time. Programmers determine at declaration time where data will be located with respect to threads and private and shared space. Private variable declarations result in one instance of the variable within the private space of each thread. Shared scalar variables have affinity to thread 0. Shared array elements are distributed by default in round-robin fashion across the threads. Such default can be changed using a block size specified by the programmer, in which case the array elements are distributed by blocks of elements across the threads in round-robin fashion. Thus, the data layout can be defined in a way that reduces the overall number of remote shared memory references. Shared variables and arrays cannot have automatic storage duration.

## EXERCISES

- 2.1 List two unique features in each of the four parallel programming models: message passing, data parallel, shared memory, and distributed shared memory.
- 2.2 Two popular computer architectures are the multicomputer architecture, in which each node has a local memory with a private address space, and the NUMA (nonuniform memory access architecture), in which there is one global address space but each node holds a part of that shared system memory. Discuss the performance pros and cons of implementing each of the four programming models on each of these two architectures.
- 2.3 In addition to the shared address space of the distributed shared memory programming model, the UPC memory model includes thread-private memories. Give two reasons why including such private memories would be a good idea.
- 2.4 Why does it make sense not to have dynamic shared variables? It was also stated that shared pointers to private variables were not recommended. Why?
- 2.5 Write a program to perform  $A = B * C$ , where  $A$ ,  $B$ , and  $C$  are  $n \times n$  matrices and  $n$  is a multiple of `THREADS`.
- 2.6 Repeat Exercise 2.5, but block the  $A$  matrix by rows. Derive an expression for the number of remote accesses needed for computing each  $C$  element and compare the results with those of Exercise 2.5.



# Pointers and Arrays

The concept of pointers is a core element of the ISO C programming language and has proven to be a powerful programming tool. UPC defines a number of pointer classes that follow directly from C pointers and the UPC memory model. As in C, pointers are variables that contain the address of another variable. However, under the UPC memory model such pointers can reside in either the private space or the shared space. UPC pointers can also reference the private space or the shared space. These different variants of the UPC pointer semantics serve distinct programming scenarios. Pointers often lead to compact programs and result in efficient implementations. Pointers and array indexes are closely related in C, and they remain that way in UPC. Under UPC, pointers can be declared to have blocking factors similar to those in the case of arrays. Thus, given any array layout in the shared space, a pointer can be declared to traverse the elements of such an array in the same sequence that C pointers traverse array elements. In many cases, casting is also allowed from one pointer type to another.

## 3.1 UPC POINTERS

The syntax of pointer declarations in UPC is similar to those of C. However, with a memory model that has shared and private spaces, there are four major UPC pointer classes: private pointers pointing to the private space, private pointers pointing to the shared space, shared pointers pointing to the shared space, and shared pointers pointing to the private space (see Figure 3.1).

Consider the following pointer declarations:

```
int *p1;           // private to private
shared int *p2;   // private to shared
int *shared p3;   // shared to private
shared int *shared p4; // shared to shared
```

The first statement declares a private pointer `p1`, which points to the private space and resides in the private space; we notice that `p1` is clearly a typical C pointer. `p2`

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

Legend :

- PP – private to private
- PS – private to shared
- SP – shared to private
- SS – shared to shared

Figure 3.1 UPC Pointer Classes

is declared as a private pointer that points to the shared space; therefore, each thread has one independent and private instance of p2. p4 is a shared pointer pointing to the shared space; thus, it has one instance with affinity to thread 0. p3 is a shared pointer to the private space and therefore should be avoided, as it would defeat the purpose of having shared space visible to all threads and private data space visible only to its associated thread.

One general observation that can help us to understand these pointer class declarations quickly is to look at the relative position of the \* character in the declaration syntax. For example, \*p is a private pointer, whereas \*shared p is a shared pointer. Note that the right part of the declaration is for the pointer type, whereas the left part is for the type of the pointed to variable. So for

```
shared int *p2; // private to shared
```

the right part, \*p2, says that p2 is a private pointer, and the left part, shared int, indicates that the variable pointed to is of type shared int. Similarly, in the declaration

```
shared int *shared p4; // shared to shared
```

\*shared p4 indicates that p4 is a shared pointer, and the left part shared int indicates that the variable pointed to is a shared int.

Figure 3.2 demonstrates where the pointers of the previous four declarations are located and to where they are pointing. This figure shows that one instance of

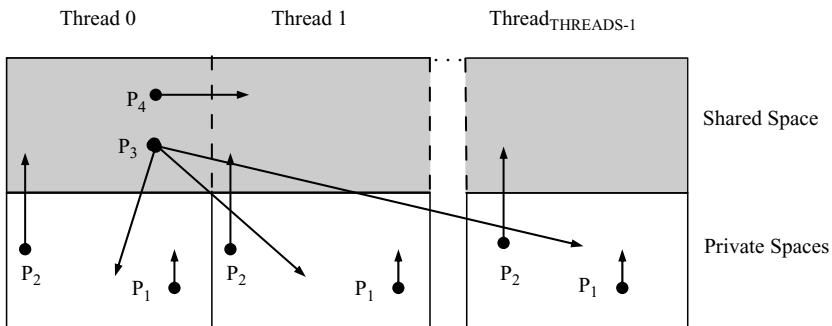


Figure 3.2 UPC Pointer Scenarios

the private pointers, `p1` and `p2`, exists for each thread. Both of these two pointers are created in the private space of every thread instance. Only one instance of `p3` and `p4` is created in the shared space. Each of the `p1` pointers points to its associated private space and can also point to the shared space that has affinity to that pointer. Each of the `p2` pointers can point anywhere in the shared data space. The pointer `p4` can also point anywhere in the shared space. As a shared pointer, `p3` has only one instance created in the shared space of thread 0. However, `p3` has no useful semantic behavior and therefore should be avoided.

### 3.2 POINTER ARITHMETIC

Pointers in UPC advance (are incremented) in a way similar to that of C pointers. When an expression is added to or subtracted from a pointer-to-shared, the pointer moves by a number of elements as given by the expression, where the elements are of the same type as the object pointed to. We first demonstrate this in cases where arrays are distributed by default, one element per thread. Then we show how pointers traverse the elements of blocked arrays. Consider, for example, the sequence of statements in Example 3.1.

**Example 3.1:** `pointer1.upc`

```
#define N 16
shared int x[N];
shared int *dp, *dp1;

dp = &x[5];
dp1 = dp + 9;
```

In this code, the first three statements define `x[]` as a shared array of 16 elements with the default distribution layout, one per thread in round-robin fashion. The third statement,

```
shared int *dp, *dp1;
```

declares both `dp` and `dp1` as private pointers to the shared array `x[]`, while the last two statements initialize `dp` to point to the element `x[5]` in array `x[]` and `dp1` to point at the ninth element after `x[5]` (i.e., element `x[14]`), shown in Figure 3.3 for the case of four threads. One thing to note here is that since `dp` and `dp1` are private pointers to the shared array, there will be one instance of `dp` and one instance of `dp1` at each of the threads. To show this more clearly, consider the code in Example 3.2.

**Example 3.2:** `pointer2.upc`

```
#define N 16
shared int x[N];
```

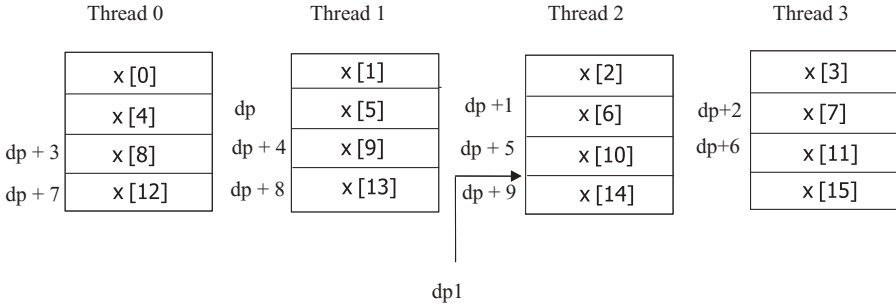


Figure 3.3

```
shared int *shared dp;
shared int *dp1;

dp = & x [5];
dp1 = x+MYTHREAD ;
```

In this code, `dp` is now a shared pointer to the shared array, so only one instance of `dp` would be created, whereas `dp1` is a private pointer to the shared data space, so each thread would have an independent instance of `dp1`. The code sets each instance of `dp1` at each thread to point at the element with the offset of `MYTHREAD` from the beginning of the `x []` array. Thus, although all threads have access to the pointer `dp`, each thread accesses only its own private `dp1`. If the code were to execute on four threads, the scenario shown in Figure 3.4 would result. The same scenario could also have resulted by replacing

```
dp1 = x+MYTHREAD;
```

with

```
dp1 = &x [MYTHREAD];
```

In the latter case, each `dp1` instance is set to point at the address of its first element of the array in each thread.

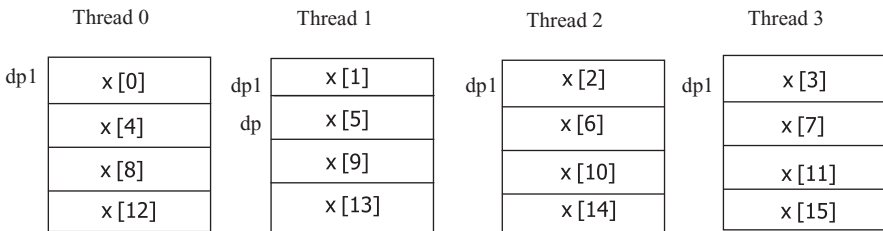


Figure 3.4



Figure 3.5 Shared Pointer Format

To keep track of shared data, a UPC shared pointer (pointer to a shared object) is composed of three fields: thread, phase, and virtual address (Figure 3.5). The thread information clearly indicates the thread affinity of the data pointed to. On the other hand, Block Address indicates the block address, and Phase indicates the location of the data item within that block. Such typical pointer implementation makes traversing blocked arrays quite easy. This also implies that to traverse the elements of a given blocked array, a pointer with the same blocking factor is needed. Consider the following example.

**Example 3.3:** pointer3.upc

```
#define N 16
shared [3] int x[N], *dp, *dp1;
shared int *dp2;
...

dp = &x[5];
dp1 = dp + 9;
dp2 = dp + 3;
```

Assuming four threads, the code in Example 3.3 would result in the layout and pointer locations shown in Figure 3.6. In Example 3.3, `dp` and `dp1` were declared with blocking factors of 3, which is also the blocking factor for the array `x[]`. These pointers point to the shared data space and view the shared space as blocks of three elements of that type. Pointer `p` was not advanced, as it was only set to `&x[5]`. This element, `&x[5]`, is in the third position of a block of 3. When `dp1` was incremented by 9 beyond the location of `p`, it immediately started with the next thread and followed the order of the array elements. This kind of blocked

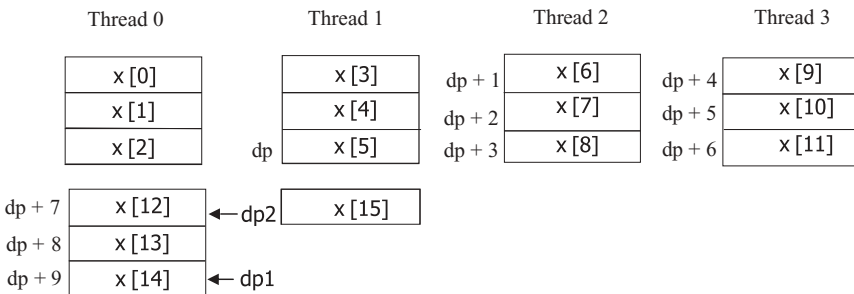


Figure 3.6



pointer is typically used to easily access the elements of blocked arrays, using the same blocking factor as the array. Pointer `p2` shows what happens when a pointer of mismatched blocking factor is used. Since no layout qualifier was specified for `p2` in its declaration, `p2` follows the default distribution regardless of how the array is laid out. Thus, `p2` moves to a new element in the next thread each time it is incremented.

### 3.3 POINTER CASTING AND USAGE PRACTICES

UPC allows the casting of one pointer type to another. However, shared pointer representations typically hold the thread and phase information (see Figure 3.5). Therefore, casting a shared pointer to a private pointer may result in the loss of the thread information. Although this is not generally advisable, it could be useful when it is desired to use a private pointer to point in its own local shared data space. On the other hand, casting a private pointer to a shared pointer is an error and would produce unknown results.

The following code demonstrates the syntax for casting a shared pointer to private data space:

```
shared int x [THREADS];
int *p;
p = (int *) &x [MYTHREAD]; /* p points to x [MYTHREAD] */
```

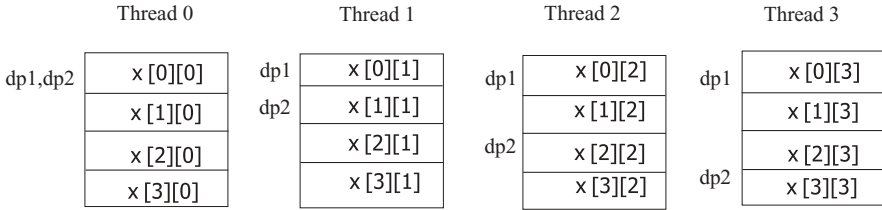
The construct `&x [MYTHREAD]` will return in each thread a different shared address pointing to its local element of `x []`. `(int *)` is a cast to private, which transforms the shared pointer to a private one. Thus, the previous statement sets each of the private pointers, `p`, to point at the `x []` element, which has affinity with its thread (i.e., **MYTHREAD**). There are three points here to reinforce: (1) local shared data is stored contiguously and can also be pointed at by private pointers; (2) a cast to private may result in losing the thread information in the shared pointer (thus, whereas casting from shared to private is allowed, the opposite is not true); and (3) as shared pointer arithmetic can be more involving than private pointer arithmetic, using private pointers instead of shared pointers when applicable may be more efficient in some implementations.

The following example shows how private and shared pointers may advance in shared data.

**Example 3.4:** pointer4.upc

```
#define N 4
shared int x [N] [N];
shared int *dp1;
int *dp2;

dp1 = x+MYTHREAD;
dp2 = (int *) dp1+MYTHREAD;
```



**Figure 3.7**

In this code, `dp1` is a set of private pointers to shared and `dp2` is a set of private pointers. In the first statement,

```
dp1 = x+MYTHREAD;
```

each of the `dp1` pointer instances will advance following the array default distribution, by **MYTHREAD** positions, from the first location in the array. Each of the `dp1` pointers will end up pointing at the first element of any array column. Each of the `dp2` pointers will be pointing **MYTHREAD** positions from the local `dp1` pointer. The private pointers, however, will not follow the order of the array elements but will advance only in their local shared space. In the case of running this code with four threads, the `dp1` pointer set will be pointing at the elements of the first row, while the `dp2` pointers will be pointing at the diagonal elements as shown in Figure 3.7.

A layout qualifier is a part of a shared pointer type. It is therefore possible to traverse the elements of a shared array with one blocking factor using a pointer with a different blocking factor. Consider the following code.

**Example 3.5:** pointer5.upc

```
shared int x [N];
shared [3] int *dp=&x [5], *dp1;
dp1 = dp + 9;
```

This code assigns to `dp1` a value that is nine positions beyond `dp`. The `dp1` pointer will follow its own blocking and not the one of the array. When executed on four threads, the pointer locations shown in Figure 3.8 result.

As a layout specifier is considered a part of a shared pointer type, casting a shared pointer with one blocking factor to a pointer that has a different blocking factor is also possible. Given the declarations

```
shared [3] int *p;
shared [5] int *q;
```

the assignment

```
p = q;
```

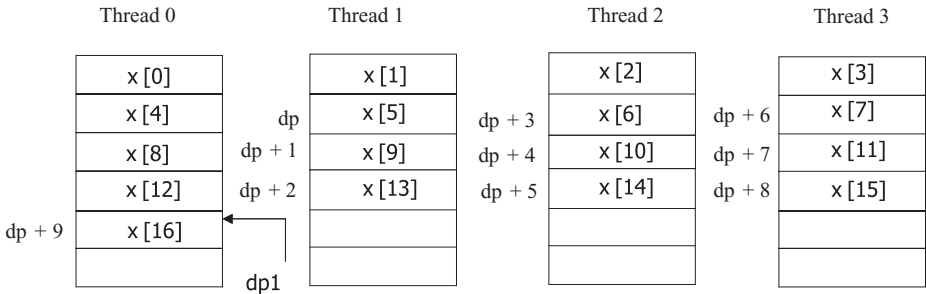


Figure 3.8

is acceptable. However, some implementations may require explicit casting in the form

```
p = (shared[3] int *)q;
```

Pointer `p` will, however, obey pointer arithmetic for blocks of three, not five. It should also be noted that pointer casting always sets the phase of the shared pointer to zero.

### 3.4 POINTER INFORMATION AND MANIPULATION FUNCTIONS

Pointer-to-shared implementations may vary from one compiler to another. However, such pointers all represent the three fields that make up such pointers-to-shared (see Figure 3.5): the thread number, the virtual address, and the phase information. Therefore, a number of special functions are included to access and manipulate pointer-to-shared information in an implementation-independent manner. These functions are often used by compiler developers, but they are available for application developers as well, if needed. UPC provides three such functions to deliver pointer information. The first,

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

returns the number of the thread that has affinity to the shared object pointed to by `ptr`. The pointer `ptr` is a private pointer-to-shared pointing to an object of any shared type. The phase of the pointer-to-shared can be also determined by a similar call:

```
size_t upc_phaseof (shared void *ptr);
```

and finally,

```
size_t upc_addrfieldof (shared void *ptr);
```

can be used in the same way to return the local address of the object pointed to by the pointer-to-shared. It should be noted, however, that the value returned by this function is implementation dependent.

In the following example these functions are used by thread 0 to print three elements of a pointer. The address field will be printed in 10 hexadecimal digits, while the thread number and the phase numbers will be printed in two decimal digits each.

**Example 3.6:** addresses.upc

```
//PrintPointerElements
#include <stdio.h>
#include <upc.h>
#include <upc_relaxed.h>

#define BLOCKING_FACTOR 3
#define SIZE BLOCKING_FACTOR* THREADS

shared [BLOCKING_FACTOR] int buffer [SIZE];

int main(void)
{
    int i;
    shared [BLOCKING_FACTOR] int *buffer_ptr;
    if (MYTHREAD == 0)
    {
        buffer_ptr = buffer + 5;
        for (i=0; i<SIZE; i++, buffer_ptr++)
        {
            printf("&buffer[%d] : ", i);
            printf("THREAD: %02d \t ",
                upc_threadof(buffer_ptr));
            printf("ADDRESS: %010Xh \t ",
                upc_addrfield(buffer_ptr));
            printf("PHASE: %02d\n",
                upc_phaseof(buffer_ptr));
        }
    }
    return 0;
}
```

**The statement**

```
buffer_ptr = buffer + 5;
```

in Example 3.6 sets `buffer_ptr` to point to the last element in the second block `buffer[5]`. Therefore, the printed thread number should be 1 and the phase should be 2. The address value is implementation-dependent, however.

In addition to the functions that return the three different elements of a pointer-to-shared, the function

```
shared void *upc_resetphase (shared void *ptr);
```

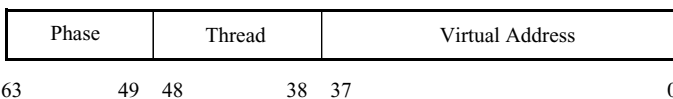
returns a pointer-to-shared which is identical to `ptr` except that it has a phase of zero. Thus, this function can be used to reset the phase of a pointer to make it point to the first element of the current block. For example, if the statement

```
buffer_ptr = upc_resetphase(buffer_ptr);
```

is used before the printing in the previous `PrintPointerFields` example, the printed phase would have been zero instead of 2.

In the rest of this section we consider a UPC pointer-to-shared implementation example (Figure 3.4) along with a possible implementation of some of the pointer information and manipulation functions. This example implementation is included for illustrative purposes only. Different [CAR99] or better implementations may be sought by compiler developers. First, assume the representation shown in Figure 3.9 for the UPC shared pointer. Then, possible definitions for the `upc_threadof()`, `upc_phaseof()`, and `upc_addrfieldof()` follow:

```
typedef uint64_t upc_shared_ptr;
size_t upc_threadof(upc_shared_ptr sh_ptr)
{
    size_t t;
    t = (sh_ptr >> 38) & 0x7FF;
    return t;
}
size_t upc_phaseof(upc_shared_ptr sh_ptr)
{
    size_t t;
    t = (sh_ptr >> 49) & 0x7FFF;
    return t;
}
size_t upc_addrfield(upc_shared_ptr sh_ptr)
{
```



**Figure 3.9** Possible UPC Pointer-to-Shared Implementation

```

size_t t;
t = sh_ptr & 0x3FFFFFFFFFULL;
return t;
}

```

In the case of `upc_threadof()`, the pointer representation is shifted to the right 38 positions to bring the thread information to the least significant part of the pointer, and the remaining bits to the left of the phase information are masked out using a bitwise logical-and operation with a mask of 15 consecutive bits, 7FF in hexadecimal. Similar shifting and masking is also used to produce the phase information, in `upc_phaseof()`, and masking only is used to generate the address in `upc_addrfieldof()`, where address information of the element in this case already occupies the lower 38 bits of the pointer representation.

A simpler but possibly less efficient implementation could use a structure with three fields to store the three elements of the pointer. The equivalent implementation of each of the three previous functions would simply amount to extracting the corresponding field from the structure.

### 3.5 MORE POINTER EXAMPLES

Consider a program for squaring a matrix  $A$ . If the matrix is shared and follows the default blocking, one can simply access the rows of the matrix using an array of pointers-to-shared. On the other hand, using the fact that local shared data is stored contiguously and can be accessed by private pointers, private pointers can be used to access the columns of the matrix. The code for this matrix-squaring example follows.

**Example 3.7:** `matsqr1.upc`

```

// Matrix Squaring
shared double A[THREADS][THREADS] , A_Sqr[THREADS][THREADS];

int main(void)
{
    shared double *rows_ptrs[THREADS]; // Points to the rows
    double        *cols_ptr; // Points to the local shared columns
    int           i, j, k;
    double        d;

    //Initialize the pointers
    for(i=0; i < THREADS; i++)
    {
        rows_ptrs[i] = &A[i][0];
    }
}

```

```

}
// Privatize (local shared to private)
cols_ptr = (double *) &A [0] [MYTHREAD];

// Matrix squaring computation
for(i=0; i<THREADS; i++)           // for each row
  upc_forall(j=0; j<THREADS; j++; j) // each thread
  { // computes one element
    d = 0;
    for(k=0; k<THREADS; k++)
      d += * (rows_ptrs [i] +k) * * (cols_ptr+k);
    A_Sqr [i] [j] = d; // write to A_sqr [row i] [col j]
  }
return 0;
}

```

In this example,

```
rows_ptrs [i] = &A [i] [0];
```

initializes the pointer `row_ptrs [i]` to point at the beginning of row `i` in array `A [] []`. Since `row_ptrs` is declared as an array of private pointers-to-shared, each thread will have its own private array of pointers that can be manipulated independently by each thread. When incremented, `rows_ptrs [i]` will advance through the elements of the `i`th row of array `A [] []`. Meanwhile, `cols_ptr` is a private pointer-to-shared. Thus, in each thread

```
cols_ptr = (double *) &A [0] [MYTHREAD];
```

casts the address of the `i`th column to private and assigns it to `cols_ptr`, so that it points at the beginning of column `i`. The cast is used since `cols_ptr` is a private pointer, whereas `&A [0] [i]` is a shared reference. When incremented, `cols_ptr` of thread `i` will now advance through the elements of the `i`th column of array `A [] []`. Since each of the threads now accesses one of the columns, the algorithm proceeds as follows. Elements of the result matrix, `A_sqr [] []`, are computed one row at a time. Thus, for each row of the matrix, each thread computes one result that goes into the column position corresponding to that thread number. Therefore, the outer loop

```
for (i=0; i<THREADS; i++)
```

sequences through the rows. Then, for each row,

```
upc_forall (j=0; j<THREADS; j++; j)
```

causes each of the threads to compute one of the elements in that row of `A_sqr [ ] [ ]`, specifically the element whose column index is equal to the thread number. The actual computations of an `A_sqr [ ] [ ]` element are performed within the body of the innermost loop.

```
for (k=0; k<THREADS; k++)
```

Instead of using an array of private pointers-to-shared for addressing the rows, one could have used a private pointer-to-shared that traverse the array elements as if it were a one-dimensional array.

**Example 3.8:** matsqr2.upc

```
// Matrix Squaring (single pointer)
shared double A [THREADS] [THREADS] , A_Sqr [THREADS] [THREADS];

int main (void)
{
    shared double *rows_ptr;
    double      *cols_ptr;
    int         i, j, k;
    double      d;

    // Initialize the pointers
    rows_ptr = &A [0] [0];
    cols_ptr = (double *) &A [0] [MYTHREAD];

    // for each row
    for (i=0; i<THREADS; i++, rows_ptr+=THREADS)
        upc_forall (j=0; j<THREADS; j++; j)      // for each local
                                                // shared column
        {
            d = 0;
            for (k=0; k<THREADS; k++)
                d += *(rows_ptr+k) * *(cols_ptr+k);
            A_Sqr [i] [j] = d;    // write to A_sqr [row i] [col j]
        }
    return 0;
}
```

As in C, pointer and array index notations may be interchanged, and pointers to matrixes can be passed to a function that may perform the matrix squaring. Making these changes to Example 3.7, the following program results.



**Example 3.9:** matsqr3.upc

```

// Matrix Squaring
shared double A[THREADS][THREADS], A_Sqr[THREADS][THREADS];
void mat_squaring(shared double (*dst)[THREADS],
                  shared double (*src)[THREADS])
{
    shared double *rows_ptrs[THREADS];
    double        *cols_ptr;
    int           i, j, k;
    double        d;

    // Initialize the pointers
    for(i=0; i<THREADS; i++)
        rows_ptrs[i] = &src[i][0];

    // Privatize (local shared to private)
    cols_ptr = (double *)&src[0][MYTHREAD];

    // Matrix squaring computation
    for(i=0; i<THREADS; i++) // for each row
        upc_forall(j=0; j<THREADS; j++; j) // each thread
        {
            d = 0;
            for(k=0; k<THREADS; k++)
                d += *(rows_ptrs[ i ]+k) * *(cols_ptr+k);
            dst[i][j] = d;
        }
}

int main(void)
{
    mat_squaring(A_Sqr, A);

    return 0;
}

```

This example touches again on an important fact: that shared objects cannot have a dynamic scope. Therefore, the shared arrays, `A[ ][ ]` and `A_sqr[ ][ ]`, are declared externally. Furthermore, only private pointers-to-shared can be declared and used inside the function itself.

### 3.6 SUMMARY

UPC embodies many powerful pointer concepts. UPC pointers include private pointers, private pointers-to-shared, and shared pointers-to-shared. Shared pointers-to-private are also possible to declare, but using them is not advised, as their behavior is not specified. Pointers-to-shared can have blocking factors, just like shared arrays. Such a layout specifier is part of the type. Pointer-to-shared representations embody three important elements: thread number, phase number, and address. Pointer information and manipulation functions can be used to extract this information from a pointer. One type of pointer can be cast into another. For example, pointers-to-shared can be cast to private pointers. In this case, the thread number information is lost. Local shared data is stored contiguously and therefore can easily be addressed by a private pointer. All these different types of pointers available in UPC can be extremely useful to developers.

### EXERCISES

- 3.1 Write a function to multiply two shared matrixes,  $A \times B$ , with the default blocking using pointers to access the elements of the matrixes. Use one single private pointer-to-shared to point at the rows of each matrix and another to point at the columns of each matrix.
- 3.2 Modify the first matrix-squaring example of Section 3.5 (Example 3.7) such that the elements of one column of `A_sqr [ ] [ ]` are computed first, then the next column, and so on. Use only private pointers-to-shared.
- 3.3 Write a function to copy the elements of a `THREADS × THREADS` matrix with a blocking factor of 5 to another with a blocking factor of 3 using blocked pointers-to-shared arithmetic.



## Work Sharing and Domain Decomposition

In this chapter we consider the distribution of workload across a number of UPC SPMD threads by decomposing the data across the threads that are executing. Distributing work across a number of threads (or processes) in a parallel application requires that each thread have the ability to identify itself through an id value and to recognize the other threads, possibly remote, available to cooperate on performing the application task. This self-identification and recognition of the other cooperating threads allows a division of labor by identifying the partition of the data that will be manipulated by each thread. In UPC, variable declarations establish the affinity of data and threads, providing locality information and control to programmers for decomposing their workload more intelligently. UPC programmers can take advantage of this knowledge by assigning each thread to apply its work, as appropriate, on the data that has affinity to that thread. In this way, the majority of the accesses can become thread local. On machines with physically distributed memory, it is expected that compilers will attempt to co-locate each thread and the data that has affinity to it onto the same physical node, thereby reducing remote accesses and improving execution time. Moreover, as in other parallel programming paradigms, UPC offers the ability for each thread to identify itself and the rest of threads available to help through the special constants **MYTHREAD** and **THREADS**, respectively, where **MYTHREAD** is in the range 0 through **THREADS**-1.

In addition to the aforementioned basic methods for work sharing, UPC features the iteration statement `upc_forall`, which distributes independent iterations across threads for parallel execution in different ways. In this chapter we cover these work-sharing concepts and their relationship to affinity of data, to show programmers how to exploit access locality. We also explore how this can be done in the context of blocked shared data structures through application examples that require interesting data distributions such as multidimensional arrays and trees.

## 4.1 BASIC WORK DISTRIBUTION

Consider a version of the matrix–vector multiplication example of Examples 2.2 and 2.3, in which a matrix of **THREADS** \*10 × **THREADS** is multiplied by a vector of length **THREADS**. Knowing the number of threads, **THREADS**, the matrix and vector dimensionalities, and given that each thread can identify itself through the special constant **MYTHREAD**, work can be distributed such that each thread operates on a unique set of rows from the matrix `a [ ] [ ]`.

*Example 4.1:* `matvect3.upc`

```
#include <upc_relaxed.h>
shared [THREADS] int a [THREADS*10] [THREADS];
shared int b [THREADS] , c [THREADS*10];
int main (void)
{
    int i, j;

    for (i = MYTHREAD; i < THREADS*10; i += THREADS)
    {
        c [i] = 0;
        for (j = 0; j < THREADS; j++)
            c [i] += a [i] [j] * b [j];
    }

    return 0;
}
```

In this example, matrix `a [ ] [ ]`, which has the dimensionality **THREADS**\*10 × **THREADS**, is multiplied by vector `b [ ]`, which is of length **THREADS**, producing vector `c [ ]`, which is of length **THREADS**\*10. The declarations

```
shared [THREADS] int a [THREADS*10] [THREADS] ;
shared int b [THREADS] , c [THREADS*10] ;
```

establish `a [ ] [ ]`, `b [ ]`, and `c [ ]` as shared arrays. Further, the `a [ ] [ ]` array is distributed in blocks of one row across the threads (i.e., each thread gets one row in round-robin fashion). At the end, each thread ends up with 10 rows in its local shared space when all **THREADS**\*10 rows are distributed.

As the loop

```
for (i=MYTHREAD; i < THREADS*10; i+=THREADS)
```

is executed by all threads concurrently in SPMD style, each thread operates at first on the row of matrix `a [ ] [ ]` that matches the thread number. Then each thread

jumps by **THREADS** row to work with its next row, and so on, until all 10 rows are processed.

In performing the operations above, it should be noted that each thread operates on the *a*-elements that are local to it only. Thus, the inherent locality in the underlying problem is fully exploited. This is generally much easier to do in UPC than in many other paradigms. Note also that one can distribute the matrix in blocks of 10 rows each. This is done by changing the matrix `a [ ] [ ]` declaration to

```
shared [THREADS*10] int a [THREADS*10] [THREADS] ;
```

The outer loop may then be slightly changed in order to continue to exploit locality and maintain the execution efficiency and good performance, but the remainder of the program remains unchanged.

## 4.2 PARALLEL ITERATIONS

In Section 4.1 we demonstrated how work can be distributed and how each thread can determine what its assigned share of work is based on **MYTHREAD**, **THREADS**, and the layout of shared data. Using these constructs and methods, it was demonstrated that a programmer can easily distribute work such that each thread can work as much as possible on data that has affinity to it, thereby maximizing data locality exploitation.

Another powerful UPC construct for specifying program parallelism is `upc_forall`, which simplifies this process and provides many options for defining workload distribution across threads. The `upc_forall` construct is a work-sharing iteration statement, which syntactically resembles the C `for` loop statement. It assumes, however, that all iterations are independent of one another and therefore can be thought of as units of work that can be distributed across the threads, where they get executed in parallel. The syntax for `upc_forall` is as follows:

```
upc_forall( expression1; expression2; expression3; affinity)
```

The first three arguments of the `upc_forall` statement, *expression1*, *expression2*, and *expression3*, are all expressions. The typical use of these is similar to that of the C `for` statement: that is, initialization, test, and update. Any of these expressions can be omitted, but the semicolons cannot be removed.

The fourth component of the `upc_forall` construct, *affinity*, is the controlling element that determines which thread executes a given iteration. Thus, *affinity* must have a unique value in each thread. Each iteration is executed by exactly one thread, but one thread can execute more than one iteration. To this end, the *affinity* component can be either an expression or a pointer-to-shared. When *affinity* is an integer expression, all iterations for which the value “*affinity* modulo **THREADS**” matches the thread number of a given thread, **MYTHREAD**, will be executed by that thread.

In other words, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value “*affinity* modulo **THREADS**.” When *affinity* is of pointer-to-shared type, a thread executes a given iteration if the object pointed to has affinity to that thread (i.e., if the object pointed to was local to that thread). In other words, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value of `upc_threadof (affinity)`.

Consider a variant of the matrix-to-vector multiplication in which the blocked matrix has a dimensionality of `THREADS*4 × THREADS*4`, and each block is of size `THREADS*4`.

**Example 4.2:** `matvect4.upc`

```
#include <upc_relaxed.h>
shared [THREADS*4] int a[THREADS*4][THREADS*4];
shared int b[THREADS*4], c[THREADS*4];

int main (void)
{
    int i, j;

    upc_forall(i = 0; i < THREADS*4; i++; i)
    {
        c[i] = 0;
        for (j = 0; j < THREADS*4; j++)
            c[i] += a[i][j] * b[j];
    }

    return 0;
}
```

In this case, the affinity expression is simply `i`. This affinity expression distributes the iterations in round-robin fashion across the threads. All iterations for which `i` modulo `THREADS` equals some integer `k` will be executed by thread number `k`, where `k` ranges from 0 to `THREADS-1`. As the `upc_forall` construct is the outer iteration statement in this example, each iteration corresponds to the processing of one row of matrix `a[][]`. The data layout selected distributes the matrix by rows, one row per thread in round-robin fashion, ensuring that all accesses to the `a[][]` matrix are local.

The foregoing scenario can be accomplished by using the affinity field as a pointer-to-shared. The only change needed is as follows.

**Example 4.3:** `matvect5.upc`

```
upc_forall(i = 0; i < THREADS*4; i++; &a[i][0])
```

The affinity field is basically a pointer to the first element of the  $i$ th row in matrix  $a$   $[[[]]$ . Alternatively,  $a[i]$  can also be used. However, some early compilers may not support this syntax, so  $\&a[i][0]$  is used.

In Example 4.3, data is distributed by blocks of one row each, in round-robin fashion. Each thread has four noncontiguous rows assigned to it. In some applications it could be better to distribute an array in groups of contiguous rows. To continue to harness the benefit of locality, equivalent chunks of iterations need to be assigned to the same thread. Consider modifying Example 4.3 to distribute data by chunks of four rows and assign every four consecutive iterations to one thread.

**Example 4.4:** `matvect6.upc`

```
#include <upc_relaxed.h>

shared [THREADS*16] int a[THREADS*4][THREADS*4];
shared int b[THREADS*4], c[THREADS*4];

void main (void)
{
    int i, j;

    upc_forall(i = 0; i < THREADS*4; i++; (i/4))
    {
        c[i] = 0;
        for (j = 0; j < THREADS*4; j++)
            c[i] += a[i][j] * b[j];
    }
}
```

The affinity is an expression in this case, which is the integer division of the current iteration number by 4, to produce one thread number for every four consecutive iterations. In addition to being an integer expression or a pointer-to-shared, the affinity field in a `upc_forall` statement can be either **continue** or left empty. In these cases, all iterations are executed by all threads. Like the ISO C `for` statement, the UPC `upc_forall` construct may be nested within the program structure either directly or through function calls. In this case the outermost `upc_forall`, which does not have `continue`, or empty affinity, is considered the controlling `upc_forall`. All of its inner `upc_forall` loops will act as if they have a `continue` or an empty affinity field, and all of their iterations will therefore be executed by all threads. The utility of these cases is quite limited. Therefore, interested readers should refer to the language specifications for more details.



### 4.3 MULTIDIMENSIONAL DATA

As an extension of ISO C, UPC stores two-dimensional data in a row-major order and maintains the ability of pointers to address multidimensional data as linear arrays. To ensure that a pointer-to-shared traverses both the elements of an array and the elements of a block in a consistent manner, data blocks are limited to be only one-dimensional. However, there are some methods, which are not very straightforward, for distributing multidimensional data blocks (discussed later). In this section we show how multidimensional data can be accessed effectively using basic UPC data distributions methods. This can be done conveniently and may even achieve better performance than distributions with traditional multidimensional blocks.

In these examples, such as Example 4.4, we distributed a matrix by groups of rows. In addition to the benefit of having a consistent way to traverse the blocks and the overall matrix using the same pointer-to-shared variable, this type of distribution can potentially limit the number of remote transactions because one-dimensional blocks create fewer ghost zones when neighboring data is needed. In general, the number of ghost zones is typically twice the number of dimensions of the block used, and the number of ghost zones typically determines the number of data transfers. The volume of remote data elements may be the same in both cases. However, most parallel architectures perform fewer transfers with more data per transfer more efficiently than do larger number of transfers with fewer data per transfer. Therefore, one-dimensional blocking lends itself to higher performance than do multidimensional distributions.

There are many ways to distribute a three-dimensional array across the threads. Figure 4.1 shows three basic approaches. Figure 4.1a shows the default case in which blocks of one element each are distributed across the threads in round-robin fashion, which can be accomplished by the following declaration:

```
shared double grid [N] [N] [N];
```

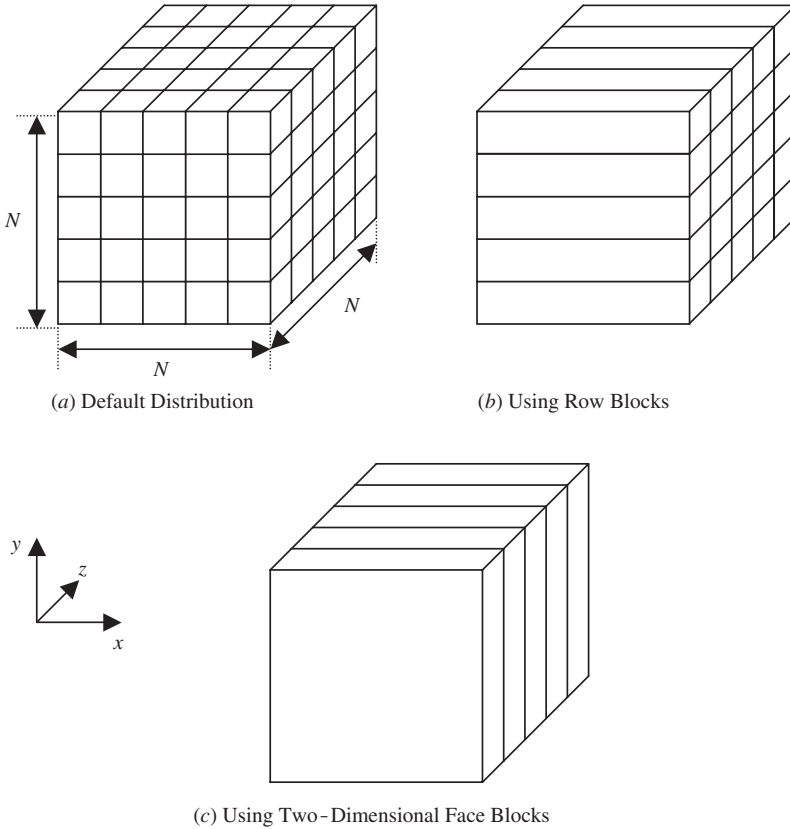
Figure 4.1b shows the case of distributing blocks of one row each, which can be established as follows:

```
shared double [N] grid [N] [N] [N];
```

and for the distribution of Figure 4.1c which distributes data by blocks of two-dimensional faces, the declaration is

```
shared double [N*N] grid [N] [N] [N];
```

Again, the preceding cases are only examples; many other declarations can be used to distribute data in other ways that are derived from the linear distribution offered by UPC, such as distributing by chunks of rows or multiple faces across the threads. The declarations are only one part of how to handle multidimensional



**Figure 4.1** Multidimensional Data and Cells

arrays and multidimensional cells. Another aspect is how to access and address the data. It is important here to note that the cell and the block do not need to be the same. The cell is simply the smallest possible entity to which application physics or computational laws are applied. The block is a data distribution unit. Although our next example will demonstrate the distributions in the previous declarations, it should be noted that having more cells per block can contribute to improved performance.

To examine multidimensional data handling in UPC, we consider the three-dimensional heat conduction problem [NTM04] in which the three-dimensional partial differential equation in a stationary medium is

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} = \frac{1}{\alpha} \frac{\partial T}{\partial t} \tag{1}$$

where  $T$  is temperature,  $t$  is time, and  $\alpha$  is the thermal diffusivity. For simplicity, we assume that the source remains hot and that its thermal properties are constant. In addition, we also assume that the entire medium is a solid with thermal diffusivity,

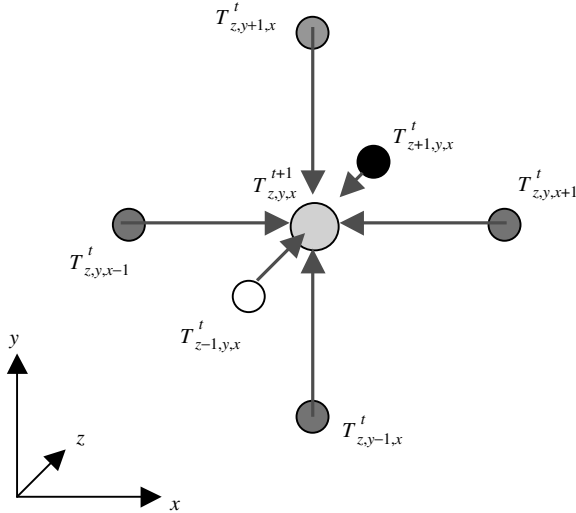


Figure 4.2 Basic Stencil Operations for the Heat Transfer Problem

$\alpha$ , of 1.0. Inside a three-dimensional cell, based on (1), the temperature is calculated as follows:

$$T_{i,j,k}^{t+1} = \frac{1}{6} \left( T_{i-1,j,k}^t + T_{i+1,j,k}^t + T_{i,j-1,k}^t + T_{i,j+1,k}^t + T_{i,j,k-1}^t + T_{i,j,k+1}^t \right) \quad (2)$$

which is the basic, stencil operation, illustrated in Figure 4.2.

The following is the code for `main()` and `initialize()`.

**Example 4.5:** `heat_conduction1.upc`

```

1. #include <stdio.h>
2. #include <math.h>
3. #include <upc_relaxed.h>
4. #include "globals.h"

5. // Declare two global grids, either source or
   // destination at
6. // each epoch, with the data distribution as described in
7. // globals.h
8. shared[BLOCKSIZE] double grids [2] [N] [N] [N];
9. shared double dTmax_local [THREADS] ;

10. void initialize(void)
11. {
12.     int y, x;
```

```

13.  /* sets one edge of the cube to 1.0 (heat) */
14.  for(y=1; y<N-1; y++)
15.  {
16.      upc_forall(x=1; x<N-1; x++; &grids[0][0][y][x])
17.      {
18.          grids[0][0][y][x] = grids[1][0][y][x] = 1.0;
19.      }
20.  }
21. }

22. int main(void)
23. {
24.     double dTmax, dT, epsilon;
25.     int finished, z, y, x, i;
26.     double T;
27.     int nr_iter;
28.     int sg, dg; // stands for source grid and destination
                // grid indexes

29.     initialize();

30.     /* sets the constants */
31.     epsilon = 0.0001;
32.     finished = 0;
33.     nr_iter = 0;
34.     sg = 0;    // using grids[0][][][] as source array
35.     dg = 1;    // using grids[1][][][] as destination array

36.     /* synchronization */
37.     upc_barrier;

38.     do
39.     {
40.         dTmax = 0.0;
41.         for(z=1; z<N-1; z++)
42.         {
43.             for(y=1; y<N-1; y++)
44.             {
45.                 upc_forall(x=1; x<N-1; x++; &grids[sg][z][y][x])
46.                 {
47.                     T = (grids[sg][z+1][y][x] + grids[sg][z-1][y][x] +
48.                         grids[sg][z][y-1][x] + grids[sg][z][y+1][x] +
49.                         grids[sg][z][y][x-1] + grids[sg][z][y][x+1]) / 6.0;
50.                     dT = T - grids[sg][z][y][x];
51.                     grids[dg][z][y][x] = T;

```

```

52.         if (dTmax < fabs (dT))
53.             dTmax = fabs (dT);
54.     }
55. }
56. }

57.     dTmax_local [MYTHREAD] = dTmax;
58.     upc_barrier;

59.     dTmax = dTmax_local [0] ;
60.     for (i=1; i<THREADS;i++)
61.         if (dTmax < dTmax_local [i])
62.             dTmax = dTmax_local [i];
63.     upc_barrier;

64.     if (dTmax < epsilon)
65.         finished = 1;
66.     else
67.     {
68.         // swapping the source and destination "pointers"
69.         dg = sg;
70.         sg = !sg;
71.     }
72.     nr_iter++;
73. } while (!finished);

74.     upc_barrier;

75.     if (MYTHREAD == 0)
76.     {
77.         printf ("%d iterations\n", nr_iter);
78.     }

79.     return 0;
80. }

```

To set up the problem size and data layout used, a `globals.h` include file is created (line 4) which could at minimum have

```
#define N 32
```

Together with the size of the array, this include file specifies one of the three data distributions depicted in Figure 4.1. This is accomplished by setting the constant `BLOCKSIZE` through a preprocessor macro definition using one of the following:

```
#define BLOCKSIZE 1
```

or

```
#define BLOCKSIZE N
```

or

```
#define BLOCKSIZE N*N
```

which would distribute data in equal blocks across all the threads. In this case two cubic grids are needed, one grid for holding the data from the preceding iteration while the new data for the current iteration is written into the other grid. To simplify the process, rather than declaring two shared three-dimensional arrays separately, line 8 declares a four-dimensional shared array, `grids`, of  $2 \times N \times N \times N$  dimensionality which can hold both arrays. The array is distributed by blocks as defined in the header file, `globals.h`. Thus, the first three-dimensional array can be indexed as `grids[0][z][y][x]`, while the second is indexed as `grids[1][z][y][x]`. The declaration is external since shared arrays cannot have dynamic scope. The variables `sg` and `dg`, declared in line 28 and initialized to 0 and 1 in lines 34 and 35, respectively, are used in the first index position to determine which of the two cubes is the source grid and which is the destination grid during a given iteration. They are also used to swap this role, of source and destination grids, at the end of each iteration according to lines 69 and 70.

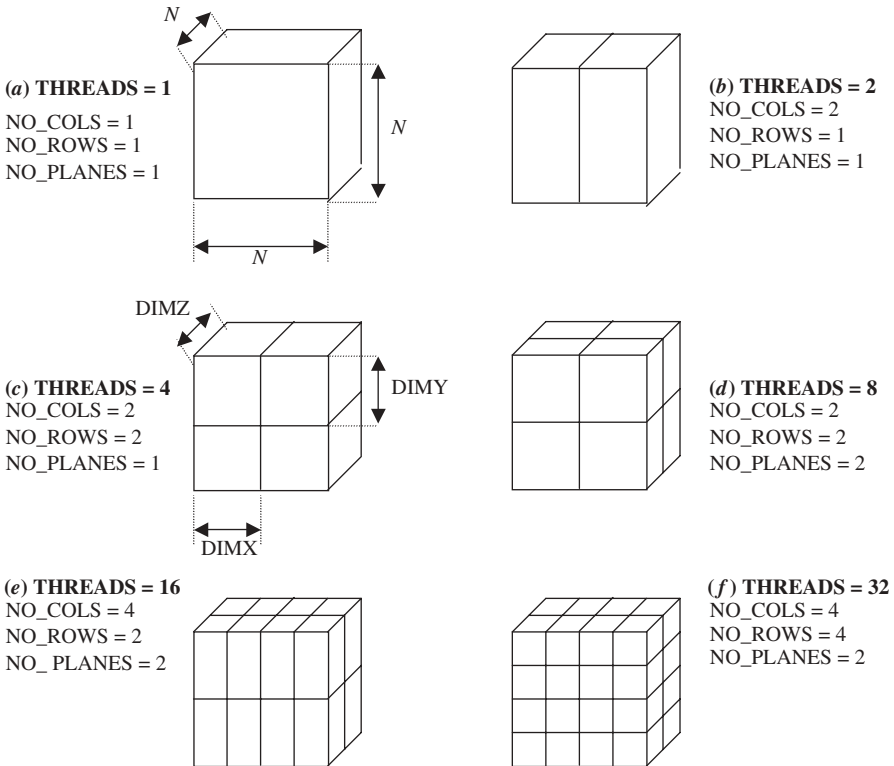
Boundary conditions are introduced at one face of each  $N \times N \times N$  grid through the function `initialize()`, lines 10 through 21. We assume in this case that  $N$  is divisible by **THREADS**. Thus, each of the threads initializes one row in surface in parallel. From the multiple assignments of line 18, this initialization is carried out for both grids simultaneously.

The simulation proceeds iteratively until the maximum change in temperature falls below a given threshold, `epsilon`, line 24. Line 24 also declares the private variables `dT` to hold the current change of temperature and `dTmax` to search for a local maximum change of temperature. The declaration in line 9 is for a shared array, `dTmax_local[]`, that has one element per thread to hold the local maximum temperature change into a shared array to simplify finding the global maximum change in temperature, lines 59 through 62. However, barrier synchronization is applied in line 58 to ensure that all threads have deposited their local maximum temperature change into that array before a global maximum is computed. The barrier present in line 63 ensures that all threads have gone through the shared `dTmax_local[]` and determined the maximum value, allowing the threads to reuse the `dTmax_local[]` buffer for the next iteration.

The bulk of work takes place between lines 38 and 73, and it is started at all threads simultaneously due to the barrier synchronization in line 37. For each  $z$  and  $y$  coordinate combination, the temperature and change in temperature are computed in parallel for all possible  $x$  coordinates using a `upc_forall` operation. From the affinity expression, each iteration is executed by the thread that has the target data point locally. If the `grids` array is distributed by the largest possible block size, most of the data accesses associated with this computation could be local. This

example demonstrates that although the array data was not distributed by three-dimensional blocks, accesses were straightforward. The reason is that the shared memory view of UPC makes dealing with the arrays similar to that of the sequential code. Meanwhile, the data distributions selected at declaration time, along with an adequate selection of the affinity field in the `upc_forall` statement, maximize locality exploitation. When all threads are done as determined by the barrier synchronization in line 74, thread 0 prints the number of iterations.

Example 4.5 demonstrated the ease and efficiency of dealing with higher-dimensional arrays in UPC. Here, we address how to partition the data into three-dimensional blocks of equal sizes across the threads, which is commonly used in message-passing codes. We then show how to modify Example 4.5 to adopt this new strategy. The discussion is limited to cases where the number of threads is a power of 2, and the number of elements in the array is divisible by the number of threads for simplicity. Examples of the target distribution are shown in Figure 4.3.



**Figure 4.3** Partitioning Using Recursive Bisection of Three-Dimensional Arrays into Equal Three-Dimensional Cells Across the Threads

Given a particular number of threads, Figure 4.3 shows how an array can be distributed such that each thread ends up with an equal-sized three-dimensional cell. The partitioning of Figure 4.3 can be determined recursively and it is commonly referred to as *recursive bisection*, where the number of cells doubles as the number of threads doubles. The partitioning for each number of threads is described uniquely by a number of rows, a number of columns, and a number of planes, as shown in Figure 4.3.

This type of data decomposition is often used in message-passing paradigms. For example, it is used in the MPI-FORTRAN version of the CG workload of the NAS Parallel Benchmark NPB-2.4 [NPB03]. There, for each power-of-2 number of threads, the grid is split in half, column-wise, row-wise, and then plane-wise, respectively, or simply across the  $x$ ,  $y$ , then  $z$  coordinates. Such grid decomposition is creating the biggest squared three-dimensional cells. Due to the static nature of the arrays, these parameters should all be computed ahead of time and inserted into a header file. In C, this domain decomposition (i.e., data partitioning parameters) can be computed as follows.

**Example 4.6:** initialize.c

```
NO_COLS = NO_ROWS = NO_PLANES = 1;
j = 0;
for (i=2; i<=THREADS; i<<=1)
{
    if ((j%3)==0)
        NO_COLS *= 2;
    else if ((j%3)==1)
        NO_ROWS *= 2;
    else
        NO_PLANES *= 2;
    j++;
}
```

After precomputing the partitioning parameters (number of rows, columns, and planes), the dimensionality of the cells in terms of array elements is determined as follows:

```
DIMX = N / NO_COLS;
DIMY = N / NO_ROWS;
DIMZ = N / NO_PLANES;
```

The values for DIMX, DIMY, and DIMZ should then be placed in `globals.h`. In fact, it is even possible to create a small program that determines the partitioning parameters and also creates the inclusion file, populating it with the dimensionality of the cells, which is the case in NPB-2.4 [NPB03].



Now consider modifying the previous heat transfer program to use this new data partitioning strategy. First, we replace the `grids[][][]` declaration.

**Example 4.7:** `heat_conduction2.upc`

```
#define CELL_SIZE DIMZ*DIMY*DIMX

struct gridcell_s {
    double cell[CELL_SIZE];
};
typedef struct gridcell_s gridcell_t;

shared gridcell_t cell_grids [2][THREADS];

#define grids(gridno, z, y, x) \
cell_grids [gridno][ ((z)/DIMZ)*NO_ROWS*NO_COLS + \
((y)/DIMY)*NO_COLS + ((x)/DIMX) ] .cell [ ((z)%DIMZ)*DIMY*DIMX + \
((y)%DIMY)*DIMX + ((x)%DIMX) ]
```

In this code, `gridcell_s` is a structure that holds all the elements of one cell in a one-dimensional array, while `cell_grids [2][THREADS]` is a two-dimensional array of two rows and **THREADS** columns, one row for the source grid and the other for the destination grid. Indexing these two virtual three-dimensional arrays properly is accomplished by the macro `grids(no, z, y, x)`. Therefore, the rest of the program should then use the macro `grids(no, z, y, x)` instead of `grids [no] [z] [y] [x]`.

## 4.4 DISTRIBUTING TREES

Trees are commonly used in a searching problem. In this section we consider distributing a tree search through an example, which considers an instance of the  $N$ -queens problem. The tree in this case will have a simple representation that is appropriate to the problem. The focus is on how to distribute the search space across the threads. Working with trees represented as linked lists is treated in Chapter 7.

### Sequential $N$ -Queens

The  $N$ -queens problem is a classic problem in computer science, due to its interesting computational behavior, depth-first searching and backtracking, and the fact that its processing time grows at a nonpolynomial (NP) rate. Thus, as the problem size grows, the execution time grows at a much more dramatic pace. There are numerous variations of the problem description. In this version of the problem, we seek to find all solutions to the problem of placing  $N$  queens on an  $N \times N$  chessboard such that no queen can kill the other. This implies that no two queens can be placed on the same row, column, or along the same diagonal.

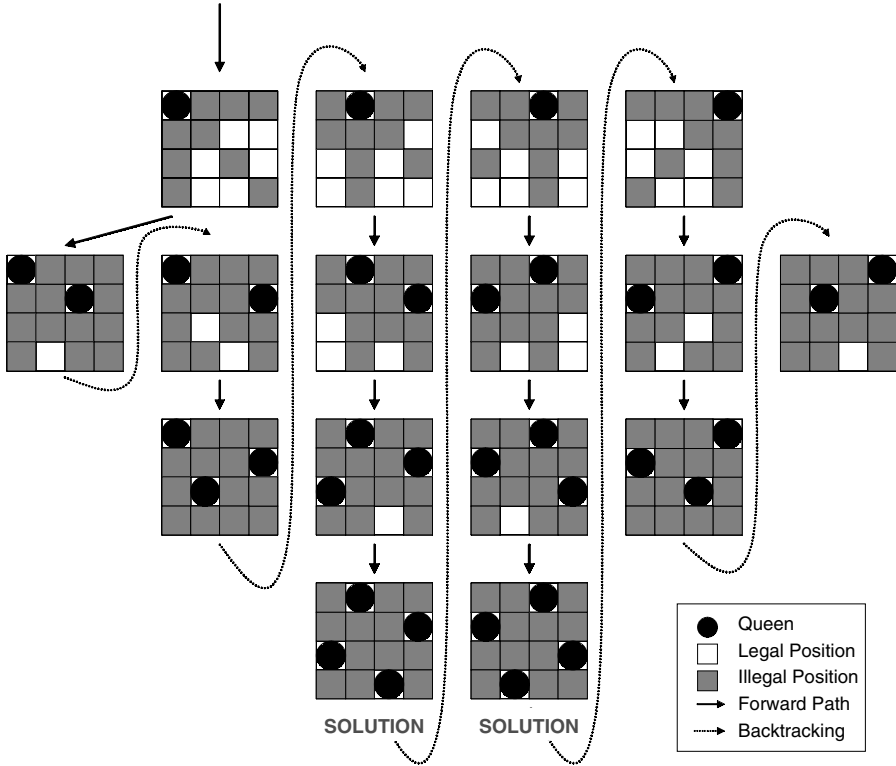


Figure 4.4 Sequential  $N$ -Queens Algorithm

Figure 4.4 shows the computational flow in the  $N$ -queens problem using a simple example of a  $4 \times 4$  chessboard. In this figure the solid arrows represent moves along a search path and the dotted lines represent backtracking after a solution was found or it was determined that the previous path was not going to lead to a solution. The solid black circles indicate queens positioned on the board. Also, a shaded box means a position that can no longer be used by a queen or it will be killed; a white box indicates an open position that can still be used. The algorithm starts by placing the first queen in the first column of the first row, then shading all boxes along the same row, column, or diagonal. Then we attempt to place the second queen in the second row in the first open position. It is clear after placing the second queen that no more queens can be placed on the board; therefore, this path was abandoned and we had to back-track to where a queen was placed in the second column of the first row. The algorithm continues, and at the end it is apparent that only two valid solutions, in which all four queens can be safely placed, exist.

One possible program implementation is to use one data structure to represent, in a binary format, the constraints associated with all prior queen placements. Another data structure can then be used to represent the new constraints that are to be created with the placement that we are about to make. Let us call these data

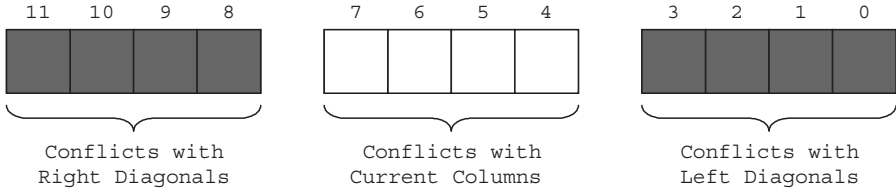


Figure 4.5 Representing Potential Conflicts

structures `vlnbl_cells`, `unsafe_cells`, and `next_row_unsafe`, respectively, and note that they will all use similar data structures. Figure 4.5 shows the data structure that can be used for `mask_t`, which is simply an integer, for the case of the  $4 \times 4$  chessboard.

The information is coded in binary, where in each binary cell a “1” means an invalid position that can cause death to the new queen and “0” means a safe position. In each 4-bit field (e.g., the conflicts with columns), the leftmost cell is used to indicate the fourth column and the rightmost indicates column 0. Figure 4.6 shows how these representations work. There are three types of information to be kept track of: the current state of unsafe cells due to queens that have already been placed (`unsafe_cells`), the vulnerable cells from which danger could be paused for a contemplated queen placement (`vlnbl_cells`), and the danger map associated with the placement of queens in the next row (`next_row_unsafe`). Figure 4.6 gives a few snapshots into the algorithm operation as it unfolds, using this binary representation of potential conflicts.

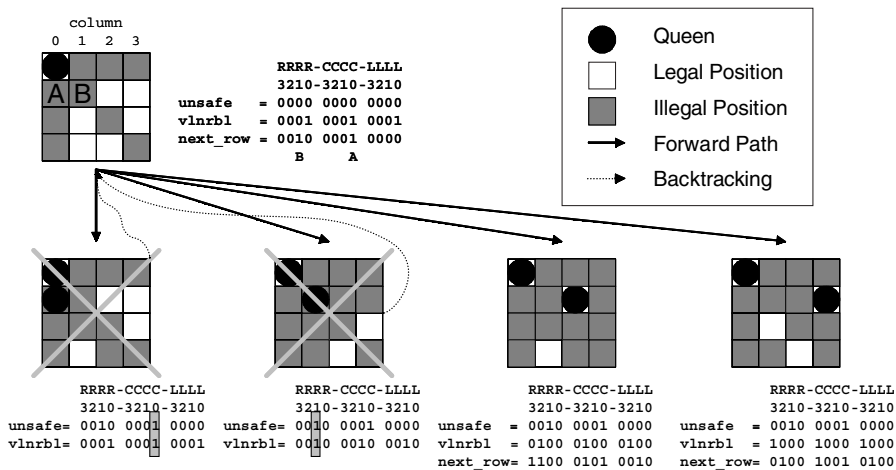


Figure 4.6 Snapshots into the  $N$ -Queens Algorithm

**Example 4.8:** nqueens1.c

```

1.  typedef uint64_t mask_t; // should be big enough for 3*N bits
2.  int N, no_solutions=0;
3.  mask_t basemask, leftdiagmask, centermask, rightdiagmask;

4.  void Nqueens(int cur_row, mask_t unsafe_cells)
5.  {
6.      int col;
7.      mask_t vrlnbl_cells, next_row_unsafe;

8.      if(cur_row == N)
9.          {
10.             no_solutions++; //Increment the number of valid
                // solutions
11.             return;
12.          }

13.     for(col=0; col<N; col++)
14.     {
15.         vrlnbl_cells = basemask<<col; // Place queen on row
16.         if(unsafe_cells& vrlnbl_cells)
17.             continue; // CONFLICT!

18.         // No conflicts here, create the unsafe_cells mask for
                // the next row
19.         // next_row_unsafe = unsafe_cells | vrlnbl_cells;
20.         // next_row_unsafe = ((next_row_unsafe&leftdiagmask)>>1
                // left
                | (next_row_unsafe&centermask) // center
                | ((next_row_unsafe&rightdiagmask)<<1); // right
                diagonals

21.         Nqueens(cur_row+1, next_row_unsafe);
22.     }
23. }

24. int main(int argc, char **argv)
25. {
26.     if(argc != 2)
27.         { printf("Usage: %s [N]\n", argv[0] ); return;}

28.     sscanf(argv[1], "%d", &N);
29.     if(N*3 > 64)
30.         { printf("64bit word not big enough!\n"); return;}

```

```

31.   basemask=1|(1<<N)|(1<<(2*N)); leftdiagmask=(1<<N)-1;
      centermask=leftdiagmask<<N;
      rightdiagmask= centermask <<N;

32.   Nqueens(0, 0);

33.   printf("Total number of solutions for N=%d: %d\n", N,
      no_solutions);
34.   return 0;
35. }

```

Example 4.8 shows a possible implementation for the sequential  $N$ -queens problem for the more general case. Lines 1 through 3 define a number of global variables, such as `basemask`, which represents the constraints that are to be created by the next queen to place. This variable should be at  $3N$  bits in size for an  $N \times N$  problem, where the columns, the left diagonal, and the right diagonal require  $N$  bits each. In addition, it defines `no_solutions`, a variable that holds the number of valid solutions at the end of the execution. Lines 4 through 23 define the `Nqueens()` function itself, and lines 24 and 25 list the `main()` driver. The `main()` obtains the size of the problem,  $N$ , from the user and sets `basemask`, based on the desired size of the problem, to 1 in the least significant bit of each of the fields: left diagonals, columns, and right columns, respectively, from right to left. The `Nqueens()` function is then called once by the `main()`. The `Nqueens()` function is recursive and will continue to call itself until it examines all solutions, returning at the end the number of valid solutions that were found, at which point `main()` prints out that number.

The function `Nqueens()`, lines 4 through 23, receives two input parameters, `cur_row` and `unsafe_cells`. Initially, it is called from `main()` with these two parameters set to 0, line 32. Thus, placement starts from the first row, with all cells on the chessboard open. Lines 8 through 12 determine whether the entire process is now completed, and if so, it terminates the program. This is done by checking if we have already reached row  $N$ , and if we have, it means that  $N$  queens have been placed on rows 0 through  $N - 1$ . Then the function increments the number of solutions and returns.

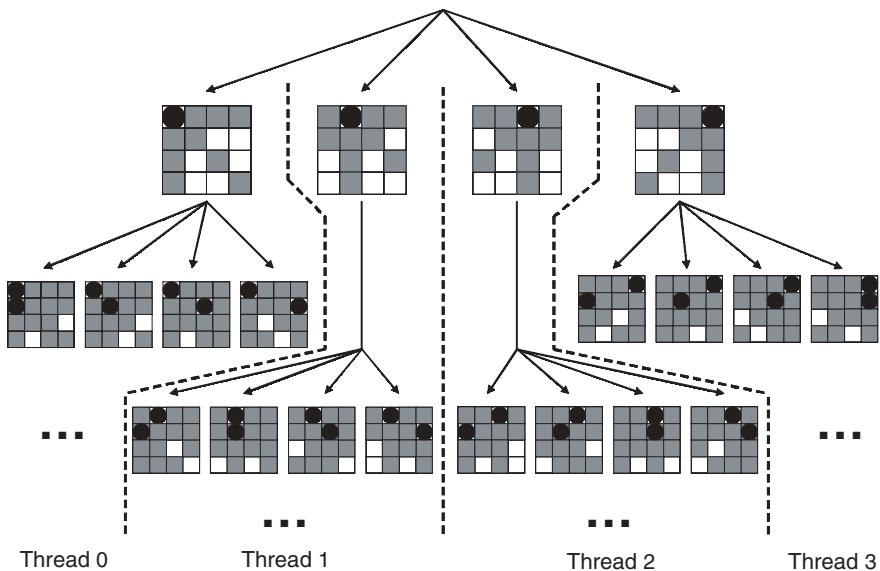
The bulk of the work takes place in lines 13 through 22. As each function invocation also specifies a current row, a `for` loop starting in line 13 simply examines all potential column placements. For each of these columns, a mask to reflect the potential exposed cells to result from this placement is created by shifting the `basemask` by the column number to the left, line 15. If these new restrictions are in conflict with the restrictions specified by the current state, as represented in `mask`, a conflict has occurred and been detected, line 17. In this case we simply skip to the next column by jumping to the next loop iteration. If there is no conflict, we compute the unsafe cells for the next row. This is accomplished by integrating the historical unsafe cells and the vulnerable cells of the current queen, then shifting

each field appropriately to reflect the threats in the next row, lines 19 and 20. The function then calls itself recursively by advancing the row in question and with a new state for unsafe cells, line 21.

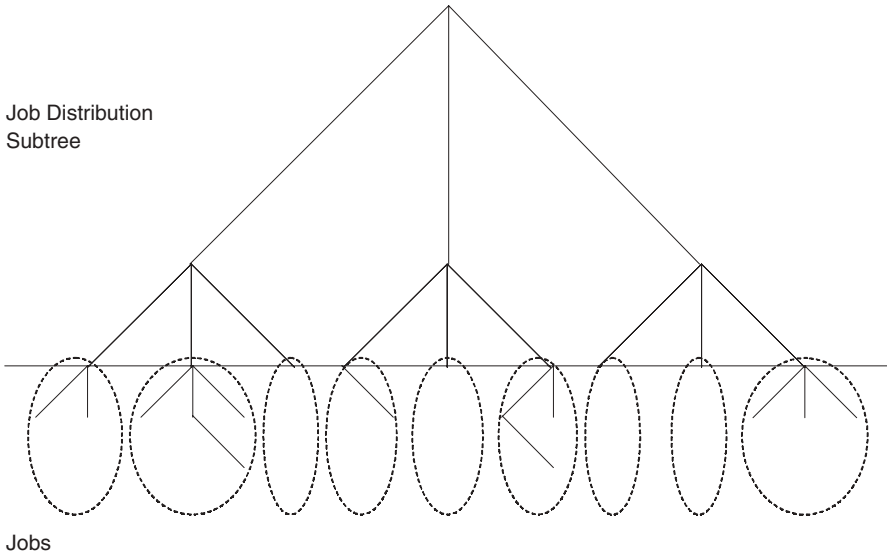
### Parallel $N$ -Queens

As the problem size increases, the number of iterations required to search for all possible ways that the  $N$  queens can coexist on the same board grows dramatically. Luckily,  $N$ -queens lends itself to parallelism. This is because  $N$ -queens is a tree-search problem in which the subtrees are independent of one another and therefore can be distributed across a number of processors easily.

Thus,  $N$ -queens can benefit from the speed of parallel processing. For example, one possibility is distributing the problem across  $N$  threads, where each thread solves the sequential problem for the case where a queen has been placed in one particular column of the first row (Figure 4.7). For larger problems and larger machines, each thread searches along one of the subtrees that correspond to a given row-column position combination in the first  $L$  rows. All threads proceed in parallel to perform a sequential search along their own subtrees. Figure 4.8 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, which means perfectly parallel with very low communication overhead.



**Figure 4.7**  $N$ -Queens Problem Using UPC

Figure 4.8 *N*-Queens Job Distribution Tree**Example 4.9:** nqueens2.upc

```

1. #include <upc_relaxed.h>
2. int N, level, no_solutions=0, method=0;
3. shared int sh_solutions[THREADS];
4. mask_t basemask, leftdiagmask, centermask, rightdiagmask;
5. #define ROUND_ROBIN 0
   ...
6. void do_job(int job, int no_jobs)
7. {
8.     int i, j, row, col;
9.     mask_t vrlnbl_cells, unsafe_cells;
10. for(j=no_jobs/N, unsafe_cells=0, row=0; j>=1; j/=N,
    row++)
11.     {
12.         col = job%(N*j)/j;
13.         vrlnbl_cells = basemask<<col;
14.         if(unsafe_cells&vrlnbl_cells)
15.             break; // CONFLICT! - ignore job
16.         unsafe_cells |= vrlnbl_cells;
17.         unsafe_cells = ((unsafe_cells&leftdiagmask)>>1)

```

```

        | (unsafe_cells&centermask)
        | ((unsafe_cells&rightdiagmask)<<1);

18.     if(j == 1)
19.         Nqueens(row+1, unsafe_cells);
20.     }
21. }

22. void distribute_work()
23. {
24.     int i, job, no_jobs;

25.     for(i=0, no_jobs=N; i<level; i++)
26.         no_jobs *= N;

27.     if(method == ROUND_ROBIN)
28.     {
29.         upc_forall(job=0; job<no_jobs; job++; job)
30.             do_job(job, no_jobs);
31.     }
32.     else
33.     {
34.         upc_forall(job=0; job<no_jobs; job++;
35.             (job*THREADS)/no_jobs)
36.             do_job(job, no_jobs);
37.     }

38. int main(int argc, char **argv)
39. {
40.     int i;

41.     if((argc != 3) && (argc != 4))
42.         {if(MYTHREAD == 0) printf("Usage: %s [N] [lvl] [CHUNK
43.             flag]\n", argv[0]); return -1;}

43.     sscanf(argv[1], "%d", &N);
44.     sscanf(argv[2], "%d", &level);
45.     if(argc == 3) method = 1; // 3 args, CHUNK method
46.                             enabled

46.     if(level >= N)
47.     {
48.         if(MYTHREAD == 0)
49.             printf("lvl should be < N\n");
50.         return -1;

```



```

51.  }
52.  if (N*3 > 64)
53.  {
54.      if (MYTHREAD == 0)
55.          printf("64bit word not big enough!\n");
56.          return -1;
57.  }

58.  basemask=1 | (1<<N) | (1<<(2*N)); leftdiagmask=(1<<N)-1;
    centermask=leftdiagmask<<N;
    rightdiagmask=centermask<<N;

59.  distribute_work();

60.  printf("TH%02d: Total number of solutions for N=%d:
        %d\n", N, no_solutions);
61.  sh_solutions[MYTHREAD] = no_solutions;
62.  upc_barrier;
63.  if (MYTHREAD == 0)
64.  {
65.      for (i=1; i<THREADS; i++)
66.          no_solutions += sh_solutions[i];
67.      printf("Total number of solutions: %d\n",
            no_solutions);
68.  }
69.  return 0;
70. }

```

In Example 4.9 the line numbers are given sequentially for ease of viewing, but be aware that “...” signifies segments of code that are not included, for the sake of brevity. Lines 2 through 4 have some of the key declarations. At line 3 an array of size THREADS is created. This is declared as shared `int sh_solutions[THREADS]` with a default distribution of [1]. This array is used to store the number of solutions that each thread discovers, where each thread stores the local number of solutions at the corresponding array cell. In addition, `level` is a row number in the chessboard that is used to determine the level at which the search tree will be segmented into independent subtrees that are to be evaluated by the various threads. The variable `method` is used to allow the user to specify whether the subtrees should be distributed across the threads by chunks of subtrees or in round-robin fashion.

Lines 46 through 51 ensure that the number of levels at which the tree is to be divided is valid. The jobs are distributed among the threads by the function `distribute_work()`, which sets the job distribution strategy as chunks or as simple round robin. The solutions are then calculated using the function `do_job()`, shown in lines 6 through 21. A barrier is placed in line 62 to

ensure that all the threads have completed finding all possible solutions. In lines 63 through 68, thread 0 aggregates the number of solutions and prints the result.

## 4.5 SUMMARY

Work sharing is simply to make each thread able to understand its responsibilities during a parallel program. Domain decomposition typically refers to partitioning the data across the threads in a manner consistent with the duties of each thread. Coming up with effective work-sharing and domain-decomposition strategies requires self-awareness, i.e., each thread should be able to identify itself, (e.g., using **MYTHREAD**) and awareness of what other threads are there to help, e.g., using **THREADS**. More convenient constructs to use are also available in UPC, such as `upc_forall()`. UPC is a locality-aware programming paradigm. Data decomposition is done primarily as a part of the declarations, and workload sharing can easily be planned in way that exploits data locality.

## EXERCISES

- 4.1 Starting from the heat conduction code, develop an edge detection image processing application using the Sobel operators.
- 4.2 Transform the  $N$ -queens problem into a maze problem. Start by developing a sequential version, then parallelize it.



## Dynamic Shared Memory Allocation

Dynamic memory allocation has been used extensively in applications for several important reasons. For example, dynamic memory allocation provides flexibility when the data sizes change during the lifetime of a program, and allows memory reuse for better utilization of the memory resources when data sizes are very large. In the private space of a UPC program, dynamic memory allocation can be done using the standard dynamic memory allocation functions available in ISO C. Dynamic memory allocation in the shared space is also possible. However, this is accomplished by special UPC memory allocation and deallocation functions. In this chapter we discuss dynamic shared memory allocation in UPC.

Dynamic shared memory allocation functions can be collective or noncollective. A function is collective if it is required to be called by all threads to produce the required impact. A function is noncollective if it should be called by a single thread. UPC has three basic dynamic shared memory allocation functions and one deallocation function. The allocation functions can create a global shared space collectively or noncollectively, or a local shared space by a thread. All dynamically allocated shared space can be released using the same deallocation function.

### 5.1 ALLOCATING A GLOBAL SHARED MEMORY SPACE COLLECTIVELY

To allocate one global shared space collectively, `upc_all_alloc()` is used. The `upc_all_alloc()` function is a *collective* function that is called by all threads using the same argument values and returns a single pointer, to the shared space allocated, to all threads. The function `upc_all_alloc()` has the following syntax:

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

where `nblocks` is the number of blocks and `nbytes` is the block size in bytes.

The shared space allocated is distributed by blocks across the threads in round-robin fashion. The `upc_all_alloc()` function takes two arguments: one specifying the number of blocks to be allocated, `nblocks`, the other specifying the size of each block in bytes, `nbytes`. The arguments are of type `size_t`, which is simply an ISO standard type definition intended to represent the maximum allocatable storage space in the target machine. The total space allocated in this case is of size `nblocks*nbytes` bytes.

Thus, the `upc_all_alloc()` function allocates shared space similar to what can be obtained from the following static declaration:

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

The `upc_all_alloc()` function returns the same pointer value on all threads. However, if `nblocks*nbytes` is zero, no memory is allocated and a null pointer-to-shared is returned. The lifetime of a dynamically allocated object extends from the time of completing the call to `upc_all_alloc()` until any thread has deallocated the object.

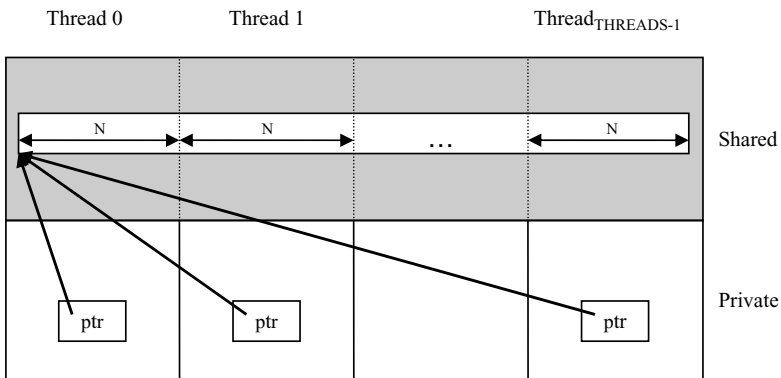
Figure 5.1 shows how `upc_all_alloc()` allocates dynamic space based on the following:

```
shared[N] int *ptr;
ptr = (shared[N] int *)
      upc_all_alloc(THREADS, N*sizeof(int));
```

The following is a more comprehensive example that demonstrates the collective allocation and use of dynamic shared space.

**Example 5.1:** `heat_conduction3.upc`

1. `#include <stdio.h>`
2. `#include <math.h>`



**Figure 5.1** Shared Space Allocation Using `upc_all_alloc`

```

3. #include <upc_relaxed.h>
4. #include "globals.h"

5. #define CHECK_MEM(var) {                                \
6.     if (var == NULL)                                    \
7.     {                                                  \
8.         printf("TH%02d: ERROR: %s == NULL\n",        \
9.             MYTHREAD, #var );                          \
10.    upc_global_exit(1);                                \
11.    } }

12. shared[BLOCKSIZE] double *sh_grids; //pointer to the
                                         grids
13. shared double dTmax_local [THREADS];
14. int initialize(shared [BLOCKSIZE] double
                 (*grids)[N][N][N])
15. {
16.     int y, x;

17.     /* sets one edge of the cube to 1.0 (heat) */
18.     for(y=1; y<N-1; y++)
19.         upc_forall( x=1; x<N-1; x++; &grids [0][0][y][x])
20.             grids [0][0][y][x] = grids [1][0][y][x] = 1.0;
21. }

22. void heat_conduction(shared [BLOCKSIZE] double
                       (*grids)[N][N][N])
23. {
24.     double dTmax, dT, epsilon;
25.     int finished, x, y, z, i;
26.     double T;
27.     int nr_iter;
28.     int sg, dg; // stands for source grid and destination
                   grid indexes
29.     /* sets the constants */
30.     epsilon = 0.0001;
31.     finished = 0;
32.     nr_iter = 0;
33.     sg = 0;      // using grids(0,,,) as source array
34.     dg = 1-sg;  // using grids(1,,,) as destination array

35.     do
36.     {
37.         dTmax = 0.0;
38.         for(z=1; z<N-1; z++)

```

```

39.     {
40.         for(y=1; y<N-1; y++)
41.         {
42.             upc_forall(x=1; x<N-1; x++; &grids[sg][z][y][x])
43.             {
44.                 T = (grids[sg][z+1][y][x] + grids[sg][z-1][y][x] +
45.                     grids[sg][z][y-1][x] + grids[sg][z][y+1][x] +
46.                     grids[sg][z][y][x-1] + grids[sg][z][y][x+1])
47.                     / 6.0;
48.                 dT = T - grids[sg][z][y][x];
49.                 grids[dg][z][y][x] = T;
50.                 if(dTmax < fabs(dT))
51.                     dTmax = fabs(dT);
52.             }
53.         }
54.         dTmax_local[MYTHREAD] = dTmax;
55.         upc_barrier;

56.         dTmax = dTmax_local[0];
57.         for(i=1; i < THREADS; i++)
58.             if(dTmax < dTmax_local[i])
59.                 dTmax = dTmax_local[i];
60.         upc_barrier;

61.         if(dTmax < epsilon)
62.             finished = 1;
63.         else
64.         {
65.             // swapping the source and destination "pointers"
66.             dg = sg;
67.             sg = 1-sg;
68.         }
69.         nr_iter++;
70.     } while(finished == 0);
71.     return nr_iter;
72. }

73. int main(void)
74. {
75.     int nr_iter;

76.     /* allocate the memory required for grids[2][N][N][N] */
77.     sh_grids = (shared[BLOCKSIZE] double *)
78.         upc_all_alloc(2*N*N*N/BLOCKSIZE, BLOCKSIZE*sizeof(double));

```

```

78.  CHECK_MEM(sh_grids);

79.  /* initialize the grids */
80.  initialize((shared[BLOCKSIZE] double (*)[N][N][N])
             sh_grids);
81.  /* synchronization */
82.  upc_barrier;

83.  /* performs the heat conduction computations */
84.  nr_iter = heat_conduction(
             (shared[BLOCKSIZE] double (*)[N][N][N]) sh_grids);
85.  /* synchronization */
86.  upc_barrier;

87.  if ( MYTHREAD == 0 )
88.      printf("%d iterations\n", nr_iter );

89.  return 0;
90. }

```

Example 5.1 gives a variation of the heat transfer problem presented in Example 4.5, which uses `upc_all_alloc()` to allocate space dynamically for the two three-dimensional grids, `grids[2][N][N][N]`. The example is also designed in a much more modular way than Example 4.5, with more functions to stress some of the points that have to do with dynamic storage allocation, pointers, and passing of array parameters.

In line 4, `BLOCKSIZE` as well as `N` are defined in `globals.h`. Lines 5 through 11 define a macro, `CHECK_MEM()`, which is used by the main program to check that memory allocation completed successfully, and if not, exits the program. Although our macro will be executed by all threads, it is sufficient to have only one thread perform this check since the allocation is done by a collective call that returns the same thing to all.

Line 12 declares a private pointer to a shared double with a block size of `BLOCKSIZE`. These pointers will be used to receive the address of the dynamically allocated space, which is to be used to hold the two three-dimensional grids. Thus, it will be viewed as a four-dimensional array of dimensionality  $2 \times N \times N \times N$ , where the first index is used to identify which of the two three-dimensional grids we are using.

Lines 14 through 21 list an initialization function that sets one face on each of the cubes to 1.0, or hot, so that they become the heat sources. This function will be called from `main()` with `sh_grids` pointer. Note that the function, however, uses array index notations for accessing its data. Alternating between pointer and array index notations is allowed in C, and the same is also valid for UPC.

Lines 22 through 72 the `heat_conduction()` function, which performs the major heat transfer iterations. It is, once again, similar to the code of Example 4.5



and included here for completeness and to demonstrate the alternation between array and pointer notations. The rest of the code is for the `main()` function. Lines 76 through 78 allocate a shared space enough to hold the two three-dimensional grids as a single four-dimensional array. The size of the space needed is sufficient to hold  $2^*N^*N^*N$  double elements. To distribute this as blocks of `BLOCKSIZE` double elements across the threads, the number of blocks is given as  $2^*N^*N^*N/BLOCKSIZE$ , and the number of bytes per element are given by `BLOCKSIZE*sizeof(double)`. Allocated space is basically one large space of bytes, and therefore a simple pointer to shared is returned. Given the way the pointer `sh_grids` was declared, assigning the returned pointer from `upc_all_alloc()` to it requires a pointer cast to (`shared[BLOCKSIZE] double*`).

The rest of `main()` then calls `CHECK_MEM()` to check whether the previous allocation was successful, and then it calls `initialize()` to initialize one face on each cube to “hot.” The `main()` then uses a barrier synchronization to ensure that all initializations are completed before calling `heat_transfer()` to perform the actual heat transfer simulation. A barrier synchronization is used after this call to make certain that all threads finished working before we print the number of iterations.

## 5.2 ALLOCATING MULTIPLE GLOBAL SPACES

One alternative allocation scheme is to use `upc_global_alloc()`, which works similar to `upc_all_alloc()`, except that it is not a collective operation. The syntax for `upc_global_alloc()` is

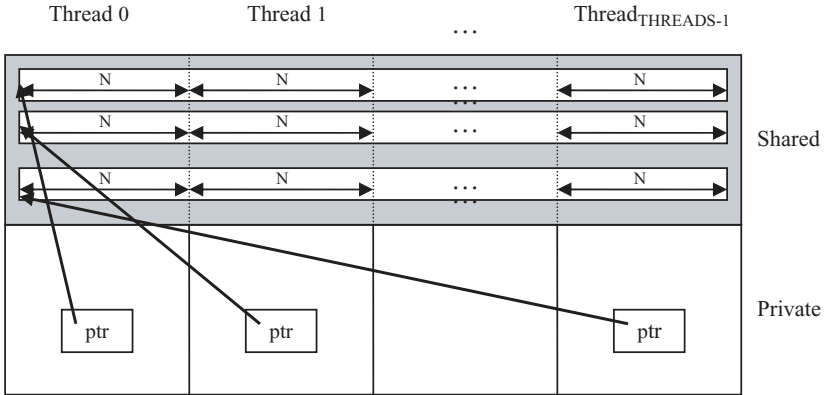
```
shared void*upc_global_alloc(size_t nblocks,
                             size_t nbytes);
```

where `nblocks` is the number of blocks and `nbytes` is the block size in bytes. In this case the space allocated is specified by the number of blocks and a block size in bytes. The function returns a pointer-to-shared, which points to the space allocated. If `nblocks*nbytes` is zero, a null pointer-to-shared is returned.

The `upc_global_alloc()` function is not a *collective* function. If called by multiple threads, all threads that make the call get different allocations. Each calling thread will then allocate a shared space that is distributed in a similar manner to the space resulting from the static declaration

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

Figures 5.2 and 5.3 show the result from `upc_global_alloc()` when called by all threads. Figure 5.2 represents how the space will be accessible if the pointer returned to shared is assigned to a private pointer to shared. This can be accomplished by



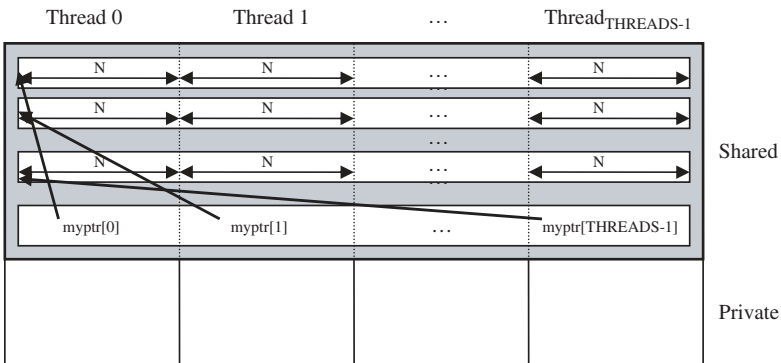
**Figure 5.2** Illustration of `upc_global_alloc`

```
shared[N] int *ptr;
ptr = (shared[N] int *) upc_global_alloc(THREADS,
    N*sizeof(int));
```

With `N` defined in the beginning of the program, this allocates a shared space of blocks of `N` integers, distributed in a round-robin way, one block per thread. There will be `THREADS` `ptr` private pointers to shared pointing at the beginning of the space allocated. These pointers will follow the pointer arithmetic of blocked pointers with a block size of `N`.

Figure 5.3 shows the case when an array of shared pointers to shared is used for accessing the space. This can be done by the code

```
shared[N] int *shared myptr[THREADS];
myptr[MYTHREAD] = (shared[N] int *)
    upc_global_alloc(THREADS, N*sizeof(int));
```



**Figure 5.3** Illustration of `upc_global_alloc` with a Shared-to-Shared Pointer

With  $N$  defined in the beginning of the program, a shared space of blocks of  $N$  integers is allocated and distributed in a round-robin way, one block per thread. There will be an array of **THREADS** shared pointer to shared, with each `myptr[MYTHREAD]` pointing at the beginning of the space allocated by that thread. These pointers will follow the pointer arithmetic of blocked pointers with a block size of  $N$ .

Example 5.2 is a more comprehensive example using `upc_global_alloc()`. It is a simple variant of Example 5.1 in which `upc_all_alloc()` is replaced by `upc_global_alloc()`.

**Example 5.2:** `heat_conduction4.upc`

```

1. ...
2. #include <upc_relaxed.h>
3. #include "globals.h"

4. // dynamically allocated sh_grids[] to be matched to
                                     // [2] [N] [N] [N]
5. shared [BLOCKSIZE] double *shared sh_grids;
6. shared double dTmax_local[THREADS];

7. int main(void)
8. {
9.     int nr_iter;

10.    if (MYTHREAD == 0)
11.    {
12.        // allocate the memory required for grids [2] [N] [N] [N]
13.        sh_grids = (shared [BLOCKSIZE] double *shared)
14.        upc_global_alloc(2*N*N*N/BLOCKSIZE, BLOCKSIZE*sizeof(double));
15.        CHECK_MEM(sh_grids);
16.    }

16.    /* synchronization */
17.    upc_barrier;

18.    /* do all the remaining, as usual */
19.    ...

20.    return 0;
21. }
```

Since `upc_global_alloc()` is noncollective and can result in multiple shared spaces if called by multiple threads, only thread 0 is making the allocation call, lines 11 through 15. Thread 0 is also checking whether the space was allocated

successfully by calling the `CHECK_MEM` macro. The rest of the code is similar to that of Example 5.1.

In modeling and simulation of linear systems, system dynamics can be represented by a matrix, while inputs to the system are vectors that need to be multiplied by the system matrix to produce an output vector. Example 5.3 considers the problem of multiplying a matrix by a sequence of vectors.

*Example 5.3:* `matvect7.upc`

```

1. #include <upc_relaxed.h>
2. #include <stdio.h>

3. shared int *a;
4. shared int *shared b[THREADS], *shared c[THREADS];

5. int main (void)
6. {
7.     shared [THREADS] int (*mat_a) [THREADS];
8.     int i, j, vect_no;

9.     a=(shared int *)upc_all_alloc(THREADS, THREADS*sizeof(int));
10.    CHECK_MEM(a);
11.    mat_a = (shared [THREADS] int (*) [THREADS]) a;
12.    b[MYTHREAD] = (shared int *shared)upc_global_alloc(THREADS,
13.    sizeof(int));
14.    CHECK_MEM(b[MYTHREAD]);
15.    c[MYTHREAD] = (shared int *shared)upc_global_alloc(THREADS,
16.    sizeof(int));
17.    CHECK_MEM(c[MYTHREAD]);

18.    /* synchronization */
19.    upc_barrier;

20.    /* initialize mat_a[ ][ ], b[ ][ ] and c[ ][ ] */
21.    ...
22.    /* synchronization */
23.    upc_barrier;

24.    /* compute the matrix by vector multiplications */
25.    for(vect_no=0; vect_no<THREADS; vect_no++)
26.    {
27.        upc_forall(i = 0; i < THREADS; i++; i)
28.        {
29.            c[vect_no][i] = 0;
30.            for (j= 0; j < THREADS; j++)

```

```

28.             c[vect_no][i] += mat_a[i][j] * b[vect_no][j];
29.         }
30.     }
31.
32.     /* synchronization */
33.     upc_barrier;
34.
35.     /* print results */
36.     ...
37.     return 0;
38. }

```

In Example 5.3, `a` is a private pointer to `shared`, which points to the matrix, while `b[ ]` and `c[ ]` are arrays of shared pointers to `shared`. Each element of `b[ ]` can be used to point to one input vector, and each element of `c[ ]` can be used to point to one result vector. Lines 9 through 13 allocate the needed memory for each one of these arrays and check to ensure that the allocation operation was performed successfully using a macro similar to the macro discussed in Example 5.1. It is noted here that `upc_all_alloc()` is used to allocate the space for the matrix. This is because only one matrix is needed. Meanwhile, since multiple vectors (in this case, **THREADS** of them are assumed for simplicity) are to be allocated in case of input and output, we use `upc_global_alloc()` to simplify the syntax.

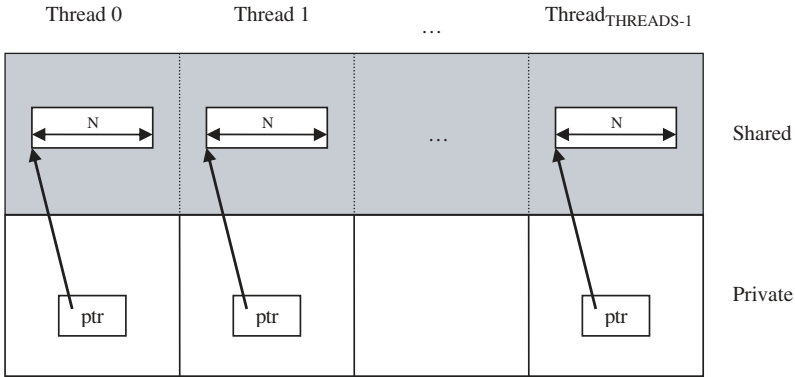
Lines 21 through 30 perform the actual computations, going through vector by vector, and for each vector number the computations are distributed by rows of `a[ ]` across the threads. This segment of code is preceded by barrier synchronization to ensure that space allocation and initializations of data structures are completed before the computation takes place. A barrier synchronization follows this block of nested loops to ensure that computations are completed before printing.

### 5.3 ALLOCATING LOCAL SHARED SPACES

In many cases it could be desirable to allocate shared but local storage. This can be done in UPC using the `upc_alloc()` function. This function takes only one argument, the number of bytes, and allocates that much shared storage with affinity to the calling thread. Thus, `upc_alloc()` is not a collective function, and it operates in a manner similar to `malloc()` in ISO C, except that the pointer returned points into the shared space. The syntax for `upc_alloc()` is as follows:

```
shared void *upc_alloc(size_t nbytes);
```

where `nbytes` is the total number of bytes to allocate. If `nbytes` is zero, the result is a null pointer-to-shared.



**Figure 5.4** Illustration of `upc_alloc`

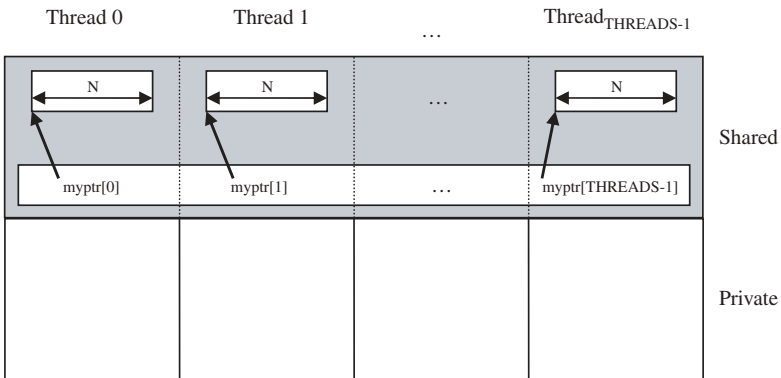
Figure 5.4 shows how `upc_alloc()` allocates and accesses space based on the statements

```
shared[] int *ptr;
ptr = (shared[] int *)upc_alloc(N*sizeof(int));
```

where `ptr` is a private pointer-to-shared with indefinite block size. The statements ultimately result in allocating `N` integers on each thread when called by all threads. Each thread will get a different pointer, and its instance of `ptr` will be pointing at the beginning of the local space allocated to it. In this case, if `ptr` is only to traverse the locally allocated memory, `ptr` does not have to be a pointer-to-shared and can be a pointer into the private space.

Figure 5.5 shows how `upc_alloc` allocates space based on the statements

```
shared[] int *shared myptr [THREADS] ;
myptr [MYTHREAD] = (shared[] int *shared)
    upc_alloc(N*sizeof(int));
```



**Figure 5.5** Illustration of `upc_alloc` with a Shared-to-Shared Pointer

where `myptr[]` is an array of **THREADS** shared pointers-to-shared, with indefinite block size. The statements ultimately result in allocating  $N$  integers on each thread when called by all threads. Each thread will get a different pointer, and its element of `myptr` will be pointing at the beginning of the local space that is allocated to it.

Example 5.4 is a modified version of the heat conduction code used throughout Chapters 4 and 5. Recalling the basic distributions of three-dimensional data in Figure 4.1, in this case we are considering a distribution in which each thread gets the largest possible chunk of faces. Those data chunks will be allocated dynamically, where each thread allocates its own chunk in its local shared space. The block size will then be  $N^*N^*N/THREADS$ .

**Example 5.4:** `heat_conduction5.upc`

```

1. #include <stdio.h>
2. #include <math.h>
3. #include <upc_relaxed.h>

4. typedef struct chunk_s chunk_t;
5. struct chunk_s {
6.     shared [] double *chunk;
7. };

8. // considering the blocksize as the biggest chunk of faces
9. int N;
10. shared chunk_t sh_grids[2][THREADS];
11. shared double dTmax_local[THREADS];

12. #define grids(no,z,y,x)
    sh_grids[no][ ((z)*N*N+(y)*N+(x)) / (N*N*N/THREADS) ] .
    chunk[ ((z)*N*N+(y)*N+(x)) % (N*N*N/THREADS) ]

13. void initialize(shared chunk_t (*sh_grids)[THREADS])
14. {
15.     // grids[][][][] has to be changed to grids(,,,,)
16. }

17. int heat_conduction(shared chunk_t (*sh_grids)[THREADS])
18. {
19.     // grids[][][][] has to be changed to grids(,,,,)
20. }

21. int main(int argc, char **argv)
22. {
23.     int nr_iter, no;

```

```

24.  /* Usage */
25.  if (argc != 2)
26.  {
27.      if (MYTHREAD == 0)
28.          printf("Usage: %s [N] \n", argv[0]);
29.      return -1;
30.  }

31.  /* get N */
32.  sscanf(argv[1], "%d", &N);
33.  if ( (N*N*N) % THREADS != 0)
34.  {
35.      if (MYTHREAD == 0)
36.          printf("N*N*N should be divisible by THREADS\n");
37.      return -2;
38.  }

39.  /* allocate */
40.  for (no=0; no<2; no++)
41.  {
42.      sh_grids[no][MYTHREAD].chunk = (shared [] double *)
upc_alloc( N*N*N/THREADS*sizeof(double));
43.      CHECK_MEM(sh_grids[no][MYTHREAD].chunk);
44.  }
45.  /* initialize the grids */
46.  initialize(sh_grids);

47.  /* synchronization */
48.  upc_barrier;

49.  /* performs the heat conduction computation */
50.  nr_iter = heat_conduction(sh_grids);

51.  /* synchronization */
52.  upc_barrier;

53.  if (MYTHREAD == 0)
54.      printf("%d iterations\n", nr_iter);
55.  return 0;
56.  }

```

Instead of giving a line-by-line explanation of Example 5.4, we will discuss only the lines that are interesting or not readily intuited. Lines 4 through 7 define a private-to-shared pointer type with indefinite blocking. In other words, when incremented, it traverses the local shared space. The intent is to have data structures of that type holding the local chunks. Lines 9 and 10 do exactly that; they



create a shared array of  $2 \times \text{THREADS}$  pointers of type `chunk_t`. Each row of these pointers is used to access one of the two three-dimensional grids, needed as the source and destination of a given iteration. In line 12 the macro `grids()`, used to simplify the indexing of the three-dimensional grids, is slightly different from its previous instances and reflects the distribution of each of the two three-dimensional grids by the largest chunk of faces, where each chunk is  $N \times N \times N / \text{THREADS}$ . Lines 24 through 38 modify the previous implementations of this example to allow the size of the problem,  $N$ , to be specified by the user interactively. This can now be afforded for two reasons. The first is clearly the use of dynamic shared memory allocation. The second is the fact that `upc_all_alloc()` and `upc_global_alloc()` require specifying both the block size and the number of blocks to allocate. To create a particular distribution, the block size must be a function of the problem size,  $N$ , which we would like to allow the user to specify at run time. However, UPC requires that the block size be a constant. In the case of `upc_alloc()`, the block size of the space allocated is always the indefinite block size. In our example, `chunk` always has an indefinite block size, and `sh_grids` has the default block size of 1.

Figure 5.6 shows the scalar ITPACK notation for compressed sparse matrixes. The nonzero elements are stored in the array `values`, the column indexes of the nonzero elements in the matrix are stored in the `colind[]` array, and `rowptr` expresses where and how many nonzero elements there are in each row.

Example 5.5 shows a UPC program to compress a sparse matrix. The example addresses the allocation and use of unequal blocks across the threads.

Sparse Matrix [NAS97]:

$$A = \begin{bmatrix} 4 & 0 & 5 & 0 & 0 \\ 6 & 3 & 0 & 7 & 0 \\ 0 & 8 & 2 & 0 & 1 \\ 9 & 0 & 0 & 3 & 0 \\ 0 & 7 & 0 & 0 & 5 \end{bmatrix}$$

Its representation using row-compressed format:

```
colind[nz] = [0, 2, 0, 1, 3, 1, 2, 4, 0, 3, 1, 4]
values[nz] = [4, 5, 6, 3, 7, 8, 2, 1, 9, 3, 7, 5]
rowptr[N+1] = [0, 2, 5, 8, 10, 12]
```

with `nz` = number of nonzero elements = 12  
`N` = dimension of  $A = 5$

**Figure 5.6** Matrix Compression Using Scalar ITPACK Format

**Example 5.5:** matcompress.upc

```

1. #include <stdio.h>
2. #include <upc_relaxed.h>

3. #define N 128

4. /* matrix to be compressed */
5. shared[N*N/THREADS] double data_A[N][N];

6. /* same matrix in its sparse representation */
7. typedef struct sparse_mat_s sparse_mat_t;
8. struct sparse_mat_s {
9.     int no_nz;
10.    shared[] int *colind;
11.    shared[] int *rowptr;
12.    shared[] double *values;
13. };

14. shared sparse_mat_t sparse_data_A[THREADS];

15. void initialize_A(void)
16. {
17.     int i, j;

18.     upc_forall(i=0; i<N; i++; i*THREADS/N)
19.         for(j=0; j<i/2; j++)
20.             data_A[i][j] = (i-j)/(N*1.0);
21. }

22. void compress_A(void)
23. {
24.     int ind, i, j;
25.     shared[] int *rowptr_ptr;

26.     /* step 1: count the number of NZ */
27.     upc_forall(i=0, sparse_data_A[MYTHREAD].no_nz=0; i<N;
28. i++; i*THREADS/N)
29.         for(j=0; j<N; j++)
30.             if(data_A[i][j] > 1E-6)
31.                 sparse_data_A[MYTHREAD].no_nz ++;

31.     /* step 2: allocate colind, rowind and values arrays */
32.     sparse_data_A[MYTHREAD].colind = (shared[] int *)
33.     upc_alloc(sparse_data_A[MYTHREAD].no_nz* sizeof(int));
33.     CHECK_MEM(sparse_data_A[MYTHREAD].colind);

```

```

34.  sparse_data_A [MYTHREAD] .rowptr = (shared [ ] int *)
    upc_alloc(N+1 * sizeof(int));
35.  CHECK_MEM(sparse_data_A [MYTHREAD] .rowptr);
36.  sparse_data_A [MYTHREAD] .values = (shared [ ] double *)
    upc_alloc(sparse_data_A [MYTHREAD] .no_nz * sizeof(double));
37.  CHECK_MEM(sparse_data_A [MYTHREAD] .values);

38.  /* step 3: create the sparse representation */
39.  rowptr_ptr = sparse_data_A [MYTHREAD] .rowptr;
40.  upc_forall(i=0, ind=0; i<N; i++, rowptr_ptr++; i* THREADS/N)
41.  {
42.      *rowptr_ptr = i;
43.      for(j=0; j<N; j++)
44.          if(data_A[i][j] > 1E-6)
45.              {
46.                  sparse_data_A [MYTHREAD] .colind[ind] = j;
47.                  sparse_data_A [MYTHREAD] .values[ind] = data_A[i][j];
48.                  ind++;
49.              }
50.  }
51.  *rowptr_ptr = ind;
52.  }

53.  int main(void)
54.  {
55.      initialize_A();

56.      upc_barrier;

57.      compress_A();

58.      return 0;
59.  }

```

Lines 4 through 13 create the data structures necessary for the input data as well as the compressed data. The original matrix data is held in `data_A[ ][ ]`, an  $N \times N$  shared matrix distributed by the largest possible chunk of rows across the threads. A structure holding the compressed matrix array values and indexes is declared, with each index and its associated value declared as private pointers to shared, with indefinite block size. A fourth field of the structure is used to hold the number of zero elements available locally. Thus, each thread will have one instance of these three pointers that can easily be used to access this information locally. The space for the nonzero values and the indexes can be allocated dynamically to be just big enough to accommodate the local needs, as the number of zero elements per chunk can vary.

Lines 15 through 21 initialize the array data. The initialization sets the nonzero elements in a lower triangle such that the base is only one-half of the matrix dimensionality. The function `compress_A()` extends from lines 22 through 52 and performs the bulk of the work. Lines 26 through 30 use a `upc_forall()` to find out how many nonzero elements each thread has in its chunk. The affinity field is selected such that each thread computes that for its own chunk in order to exploit locality.

Lines 31 through 37 allocate the memory space to hold the values and the indexes for the new representation. The space allocated for each thread is only enough to hold these data for the local chunk, based on the local nonzero element count that was found earlier from lines 26 through 30. Lines 38 through to the end of the function perform the operations needed to compute the elements that go into the nonzero values and the index arrays. The main program follows and calls the previously mentioned functions to initialize and then wait on a barrier to ensure that all initializations took place before the compression took place.

## 5.4 FREEING ALLOCATED SPACES

The benefit from dynamic memory allocation would not be complete without being able to deallocate the space when it is no longer needed. This is one of the main distinctions between dynamic and static memory allocation. In many cases, when reusing the space is not an issue, one may simply do nothing, and at the program exit, all allocated space is freed. However, for large applications where reusing space is important, `upc_free()` can be used. This call has the following syntax:

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

The `upc_free()` function frees the dynamically allocated shared storage pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer returned earlier by the `upc_alloc()`, `upc_global_alloc()`, `upc_all_alloc()`, or `upc_local_alloc()` functions, or if the space has been deallocated by a previous call, by any thread, to `upc_free()`, the behavior is undefined.

In many of the previous programming examples, we could have used `upc_free()`. For example, in Example 5.1 we can use

```
upc_free(sh_grids);
```

between lines 87 and 88. Similarly, for Example 5.5 we could have used

```
upc_free(sparse_data_A[MYTHREAD].colind);
upc_free(sparse_data_A[MYTHREAD].rowptr);
upc_free(sparse_data_A[MYTHREAD].values);
```

between lines 57 and 58.

There are many intriguing examples that require changing the size of the memory used, such as iterative data compression (e.g., using the Wavelet transform, where data gets smaller after each iteration). Memory allocation and freeing can work nicely in these cases.

## 5.5 SUMMARY

In this chapter we discussed the three different ways to allocate shared storage space under UPC. The function `upc_all_alloc()` is a collective function that is called by all threads and allocates one shared space and returns one pointer to the space allocated to the threads. The function `upc_global_alloc()` is a non-collective version of the `upc_all_alloc()`. Each calling thread gets a separate shared space and is returned a different shared pointer to that space. Finally, `upc_alloc()` is a noncollective function. When called, each calling thread gets a pointer to a shared space with affinity to the calling thread, and a different pointer is returned to each calling thread. To free the space allocated previously, one can use `upc_free()`.

## EXERCISES

- 5.1 Improve the matrix-squaring example to accept any  $N$  dimension, as specified at the command line [using `upc_all_alloc()`].
- 5.2 Enhance the vector addition example to allow the user to enter  $N$  and the values for vectors `v1` and `v2` interactively [using `upc_global_alloc()`].
- 5.3 Develop an  $N \times N$  matrix multiplication with  $N$  given at the command line, using `N*N/THREADS` as the block size [using `upc_alloc()`].
- 5.4 Implement the three-dimensional cell variant of the heat conduction problem, accepting  $N$  at the command line [using `upc_alloc()`].

# Synchronization and Memory Consistency

Although parallel processing enables fast execution, it also creates the need for synchronization. Synchronization operations ensure that although threads execute independently, their efforts are coordinated such that the same result that would be produced by a sequential execution is achieved. It is, however, equally important not to overspecify such coordination, in order to give threads enough flexibility to overlap their activities and optimize the overall execution time. Synchronization is in general one of the more difficult issues in parallel computing. However, it should be noted that for most scientific codes, the needed synchronization concepts and constructs are typically the simpler ones, and Sections 6.1 and 6.2 might be providing all or most of what is needed for scientific applications. Software system applications (e.g., a parallel database management system) typically require the more advanced concepts. UPC supports a range of synchronization primitives from the simple to the sophisticated to support all application needs. Therefore, in addition to scientific applications, we consider a number of standard computer science problems that can illustrate some of the UPC concepts in this chapter, such as the dining philosophers and the bakery shop.

One can differentiate between control synchronization and data (memory consistency) synchronization. Along the lines of control synchronization, two questions that can be asked are:

1. What if the next part of the code, due to some dependence, should not be executed until all threads finish the work at hand?
2. What if a part of the code should be executed on a mutually exclusive basis (i.e., it must be processed atomically by no more than one thread at any given point of time)?

To address these questions, UPC provides barrier synchronization and locks, which will be explained later. On the other hand, from a data synchronization perspective, two of the questions that can be asked are:

1. If a shared variable is written by a thread, when should the modified version become visible by other threads?
2. For a sequence of writes into the shared space, is it critical to make the writes in the order specified, or should we allow the system to reorder them for efficiency?

To support these, UPC provides fences and other memory consistency controls that are explained later in the chapter.

### 6.1 BARRIERS

One of the first synchronization mechanisms that comes to mind in parallel computing is *barrier synchronization*. UPC offers different types of barrier synchronization. The syntax for `upc_barrier` is

```
upc_barrier expressionopt
```

where `expression` is an integer expression. The integer expression is optional and can be used as a mark to tag a particular barrier, for debugging purposes. This can be useful in case more than one barrier can be active concurrently by perhaps two different groups of threads. However, the integer expression is not required even in such cases.

Figure 6.1 explains the operation of a barrier. All threads must arrive at the `upc_barrier` before any of them can proceed any further in the code. The barrier

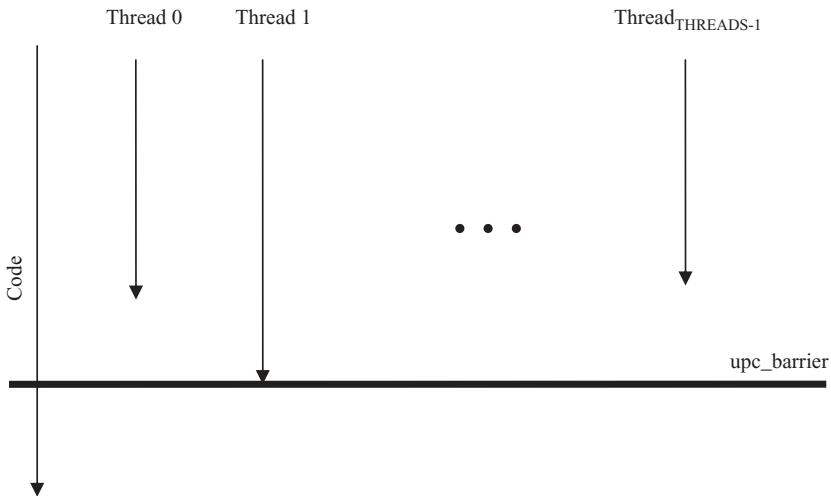


Figure 6.1 Barrier

concept has been used in many of our previous examples. Example 6.1 is taken from Example 4.5 with most statements removed and focus on the synchronization side.

**Example 6.1:** heat\_conduction1.upc

```

81. ...
82. shared double dTmax_local [THREADS];
83. ...
84. int main(void)
85. {
86.     ...
87.     initialize();
88.     upc_barrier; /* synchronization */
89.     do {
90.         // process all the local data
91.         dTmax_local [MYTHREAD] = dTmax;
92.         upc_barrier;

93.         // perform a reduction operation
94.         dTmax = dTmax_local [0];
95.         for(i=1; i<THREADS; i++)
96.             if(dTmax < dTmax_local [i])
97.                 dTmax = dTmax_local [i];
98.         upc_barrier;

99.         if(dTmax < epsilon)
100.            finished = 1;
101.         else
102.            // swapping the source and destination "pointers"
103.            nr_iter++;
104.     } while (!finished);

105.     // the processing is done here
106.     upc_barrier;

107.     // print results and exit
108. }

```

In Example 6.1, a barrier in line 88 (of Example 4.5) ensures that synchronization is complete before heat transfer processing. This is essential here since operations at the boundary of the local data of one thread will require data from the neighboring thread(s). This creates a producer–consumer scenario in which the consuming thread should ensure that the correct data has been made available by the producing thread before attempting to use it. If, however, these were an embarrassingly parallel computation in which each thread initializes its own data,



then operates on it with no need for remote accesses, the barrier would not have been necessary.

Line 92 has another barrier, which is used prior to a reduction operation. Prior to the barrier, each thread deposits its maximum change in temperature to its local element of the shared array `dTmax_local`. When all threads have done so, which is now guaranteed by `upc_barrier`, the reduction is to search for the global maximum change in temperature in `dTmax_local`. All threads do that concurrently, ending with the same maximum in their local instance of `dTmax`. The barrier in line 106 ensures that all threads have completed their search for the global maximum of `dTmax` before going to the next iteration. The last barrier is just after the processing is completed and before the printing of the results.

## 6.2 SPLIT-PHASE BARRIERS

In general, synchronization is an overhead that causes threads to wait, wasting machine cycles that could have been directed to useful processing. To reduce the performance impact of synchronization, the concept of split-phase (or nonblocking) barriers has been introduced into UPC. The idea is that if a thread completes the work needed for the next part of processing, then rather than waiting, it can perform local processing that does not affect the other threads until they are all ready to proceed to the next phase of the computation. This can reduce the idle time and improve efficiency.

Figure 6.2 illustrates the split-phase barrier operations. Once a thread finishes the computations that required the synchronization, it executes a `upc_notify` to

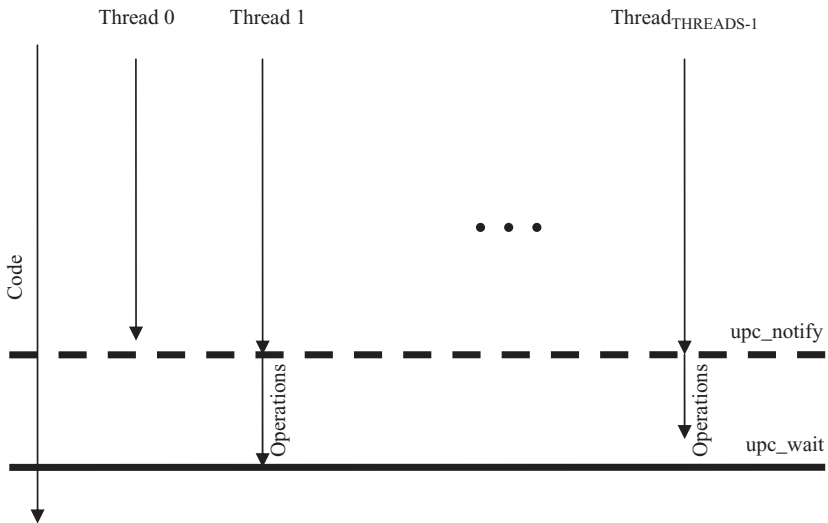


Figure 6.2 Split-Phase Barrier

inform the other threads of its standing. It can then turn its attention to local computations, and when they are also completed, the thread will execute a `upc_wait` to stay in place until all others have executed the `upc_notify`. When all other threads have executed their `upc_notify`, the thread that has executed the `upc_wait` becomes free to move to the next phase of computations without waiting for the others. The syntax for the `upc_notify` and `upc_wait` is as follows:

```
upc_notify expressionopt;
upc_wait expressionopt;
```

where `expression` is an integer expression.

`upc_wait` and `upc_notify` are collective operations that are executed by all threads. The expression, again, can be used for tagging the barriers for debugging purposes; otherwise, the expression is not required. To simplify the discussion, one can define *synchronization phase* as the execution of all statements between the completion of one `upc_wait` and the start of the next. The `upc_wait` statement will generate a run-time error if the value of its expression does not equal the value of the expression by any of the `upc_notify` statement for the current synchronization phase. No error will be generated if either statement does not have an expression.

**Example 6.2:** `heat_conduction6.upc`

```
1. #define BLOCKSIZE N*N*N/THREADS
2. // statically allocate grids[] to be matched to [3][N][N][N]
3. shared[BLOCKSIZE] double grids[3][N][N][N];
4. shared int finished[THREADS], nr_iter[THREADS];

5. void process_face(int z, int sg, int dg, double *dTmax)
6. {
7.     double T, dT;
8.     int y, x;

9.     for(y=1; y<N-1; y++)
10.    {
11.        for(x=1; x<N-1; x++)
12.        {
13.            T=(grids[sg][z+1][y][x]+grids[sg][z-1][y][x]+
14.              grids[sg][z][y-1][x]+grids[sg][z][y+1][x]+
15.              grids[sg][z][y][x-1]+grids[sg][z][y][x+1])/6.0;
16.            dT = T - grids[sg][z][y][x];
17.            grids[dg][z][y][x] = T;

18.            if(*dTmax < fabs(dT))
19.                *dTmax = fabs(dT);
20.        }
21.    }
```

```

19.     }
20. }

21. void initialize(void)
22. {
23.     finished[MYTHREAD]= nr_iter [MYTHREAD]= 0;
24.     ...
25. }

26. int heat_conduction(void)
27. {
28.     // Initialize the vars as usual
29.     ...

29.     dTmax_local[MYTHREAD] = epsilon;

30.     do
31.     {
32.         dTmax = 0.0;

33.         if((N/THREADS) == 1)
34.         {
35.             // only one face per thread
36.             if((MYTHREAD != 0) && (MYTHREAD != (THREADS-1)) &&
37.                (finished[MYTHREAD] == 0))
38.                 process_face(MYTHREAD, sg, dg, &dTmax);
39.             upc_notify nr_iter[MYTHREAD];
40.         }
41.         else
42.         {
43.             // compute front and back faces then NOTIFY
44.             if((MYTHREAD != 0) && (finished[MYTHREAD] == 0))
45.                 process_face((N/THREADS)*MYTHREAD, sg, dg, &dTmax);
46.             if((MYTHREAD != (THREADS-1)) && (finished[MYTHREAD] == 0))
47.                 process_face(((N/THREADS)*(MYTHREAD+1))-1, sg, dg,
48.                               &dTmax);
49.             upc_notify nr_iter[MYTHREAD];

50.             // compute interior faces
51.             for(z=((N/THREADS)*MYTHREAD)+1;
52.                (z<((N/THREADS)*(MYTHREAD+1)-1)) &&
53.                (finished[MYTHREAD] == 0); z++)
54.                 process_face(z, sg, dg, &dTmax);
55.         }

```

```

52.     upc_wait nr_iter[MYTHREAD];

53.     if(finished[MYTHREAD] == 0)
54.     { // normal execution -- nobody has yet reported that is the
        // end
55.         for(i=0; (i<THREADS) &&
            (dTmax_local[i]<=dTmax_local[MYTHREAD]); i++);

56.         if((i == THREADS) && (dTmax_local[MYTHREAD]<epsilon))
57.         {
58.             // someone reports that this is the end !!!
59.             upc_notify nr_iter[MYTHREAD]+1; // to match the notify
                // done by nonfinished threads

60.             no = 1; // let all threads stop at the same iteration
61.             finished[MYTHREAD] = 1;
62.             while(no != THREADS)
63.             for(i=0; i<THREADS; i++)
64.                 if((finished[i] == 0) &&
65.                    (nr_iter[i] > nr_iter[MYTHREAD]))
66.                 {
67.                     finished[i] = 1;
68.                     no ++;
69.                 }

70.             upc_wait nr_iter[MYTHREAD]+1; // to wait for them to
                // exit the function

71.         }
72.         else
73.         {
74.             // swapping the source and destination "pointers"
75.             sg = dg;
76.             dg = dg+1;
77.             dg %= 3;

78.             // for 'nonfinishing' threads ...
79.             nr_iter[MYTHREAD] ++;
80.             dTmax_local[MYTHREAD] = dTmax;
81.         }
82.     }
83.     else
84.         nr_iter[MYTHREAD]--; // last iteration was useless. ... we did
            // not complete it

85. } while (finished[MYTHREAD] == 0);
86. return nr_iter[MYTHREAD];
87. }

```

Example 6.2 is a modified version of the heat transfer problem which tries to take advantage of the split-phase barrier. The general idea is to start processing the boundaries first. Boundary processing is followed by a `upc_notify`. After the rest of the processing is completed, a `upc_wait` is executed. The `upc_wait` forces the executing thread to wait until the rest of the threads have reached `upc_notify`. At that point the thread can start the new iteration without having to wait for the others to reach the end of the current iteration. Thus, at any point a maximum of two iterations can partially overlap in execution. Recalling our initial implementation of this problem, a source grid and a destination grid were used to support the executing iteration. Now, with two iterations being overlapped in execution, three grids will be used to support the source and destination needs of the two executing iterations. Thus, line 2 declares `grids` to accommodate three  $N \times N \times N$  grids, with each grid distributed by the largest chunk of contiguous faces. Lines 5 through 20 are for processing a face. Note that faces are distinguished with the `z` index, where `z=0` means the front, `z=N-1` is the back face, and the rest are interior faces. Processing is done on a face-by-face basis by calling the function `process_face()`.

Line 30 is the beginning of the processing iterations. In line 33 we check for the special case in which each thread gets a chunk of exactly one face. In this case only the faces associated with threads 1 through `THREADS-2` are processed one by one, by calling the function `process_face()` by each of the threads. When a thread is done processing the corresponding face, it executes a `upc_notify` and proceeds.

Lines 43 and 44 discover the front faces for each thread and process them; lines 45 and 46 discover the back faces and process them. Line 47 then executes a `upc_notify`. Thus, in each of these boundary cases, each thread performs the respective processing of the boundary conditions, then executes a `upc_notify`.

Lines 49 through 52 process the interior faces and are followed by a `upc_wait`. Following line 53, any thread that has the finished flag set falls to line 85, where it will be exiting the loop and forcing the entire program to exit after adjusting the number of iterations. If not finished, each thread checks whether its value of `dTmax_local` is the greatest, line 55. If so and if this value is less than the threshold, as in line 56, preparations to end the program execution follow. However, in line 59, `upc_notify` is used to make sure that all threads made this check. The iteration number is used to tag `upc_notify` in this case.

Lines 60 through 70 are reached if one thread determines that it is time to end, and are therefore used to cause all other threads to exit in the current iteration. If not, processing continues as usual, lines 73 through 81.

Several constraints should be observed when using barriers and split-phase barriers. First, barrier nesting and overlapping is not permitted. Each thread should execute an alternating sequence of `upc_notify` and `upc_wait` statements, starting with `upc_notify` and ending with `upc_wait`. A `upc_wait` statement completes, and the thread enters the next synchronization phase, only after all threads have completed the `upc_notify` statement in the current synchronization phase. A null strict reference is implied before a `upc_notify` statement and after

a `upc_wait` statement. This is explained more in the context of memory consistency. For now it means that all outstanding writes to the shared space are completed before the `upc_notify` and after the `upc_wait` statements.

It should also be noted that the `upc_barrier` statement is essentially equivalent to the compound statement

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

Thus,

```
{ upc_notify;    upc_barrier;    upc_wait; }
```

will result in a run-time error, as it is equivalent to

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

and as pointed before, nesting is not permitted, and each `upc_notify` must be followed by a `upc_wait`.

### 6.3 LOCKS

Locks are mechanisms used to control access to the critical sections of code. Critical sections are those parts of the code that should not be in execution by more than one thread at any given point of time. A lock is basically a data structure that can be seen by all threads and has two states, locked and unlocked. When a thread is ready to enter that critical section of the code, it has first to obtain the lock and then enter the critical section. This amounts to entering and locking the critical section of the code such that no other thread can enter. When the thread completes execution of the critical section, it simply unlocks the lock, giving other threads the ability to enter the critical section. UPC locks must be allocated before use. A UPC lock variable is declared as type `upc_lock_t`. A lock is an opaque object, as its nature is implementation-dependent and therefore can be manipulated only through pointers. As in the case of dynamic shared memory allocation, allocating locks can be done collectively or noncollectively. To create a lock collectively, we use the function

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

where the `upc_all_lock_alloc` function allocates a lock dynamically and returns a pointer to it. The lock is created in an unlocked state. Since `upc_all_lock_alloc` is a collective function, the return value on every thread points to the same lock object.

Alternatively, lock allocation is also possible using the noncollective call:

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

where the `upc_global_lock_alloc` function allocates a lock dynamically and returns a pointer to it. The lock is created in an unlocked state. The `upc_global_lock_alloc` function, however, is not a collective function. If called by multiple threads, all threads that made the call get different locks.

Just as in the dynamic shared memory allocation, if we allocate, we should be able to deallocate. In UPC this is done by the `upc_lock_free` function, which has the form

```
void upc_lock_free(upc_lock_t *ptr);
```

The `upc_lock_free` function frees all resources associated with the dynamically allocated `upc_lock_t` pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer returned earlier by the `upc_global_lock_alloc` or `upc_all_lock_alloc` function, or if the lock has been deallocated by a previous call to `upc_lock_free`, the behavior is undefined.

In addition to allocating and deallocating locks, an executing thread should be able to lock and unlock it. These two primitives of manipulating locks follow:

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

The `upc_lock` function locks a shared variable of type `upc_lock_t`, pointed to by the pointer given as argument. If the lock is not used by another thread, the thread making the call gets the lock and the function returns. Otherwise, the function keeps trying to get access to the lock, waiting for the lock to be unlocked. All outstanding accesses to shared objects are completed after the call is made to `upc_lock()`. If the calling thread is already holding the lock referenced by `ptr`, the result is undefined.

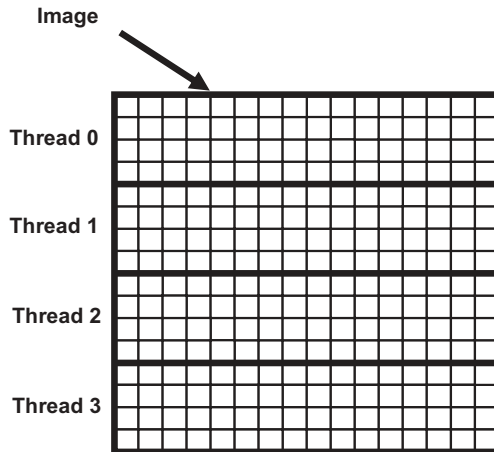
To unlock the lock when finishing a critical section, we use the unlock function:

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

The `upc_unlock` function frees the lock and does not return any value. A strict reference is implied before `upc_lock`, which means that all outstanding shared accesses are completed before the `upc_unlock()`.

One problem with `upc_lock` is that it blocks the executing thread until the lock is available, which can result in many wasted cycles. It would be much more efficient if such process blocking is avoided and a call can just check on the availability of the lock and acquire it if it is available. If not, it should just return and indicate that it did not yet acquire the lock, so that other work can be performed while the lock is unavailable. The function `upc_lock_attempt` does simply that and is described as.

```
int upc_lock_attempt(upc_lock_t *ptr);
```



**Figure 6.3** Histogram Data Decomposition

The `upc_lock_attempt` function tries to lock a shared variable of type `upc_lock_t`, pointed to by the pointer given as argument. If the lock is not held by another thread, the thread making the call gets the lock and the function returns 1. Otherwise, the function returns 0. A null strict reference, described in Section 6.4, is implied after a call to `upc_lock_attempt()`, which returns a 1.

Histogramming is commonly used in image-processing applications. As shown in Figure 6.3, given an image, which is simply a two-dimensional array of pixels (or picture elements each reflecting a gray-scale number), it is desirable to know the number of pixels in the image for each gray level. Example 6.3 shows a UPC program that solves the histogramming problem for a  $1024 \times 1024$  image with gray levels in the range 0 to 255.

**Example 6.3:** `hist1.upc`

```

1. #include <upc_relaxed.h>
2. #define N 1024
3. shared [N*N/THREADS] unsigned char img [N] [N];
4. shared int hist [256];
5. upc_lock_t *lock;
6. void initialize(void)
7. {
8.     lock = upc_all_lock_alloc();
9.     CHECK_MEM(lock);

```



```

10.     // initialize the image img[][]
11.     }

12. int main(void)
13. {
14.     int i, j;

15.     initialize();

16.     upc_barrier;

17.     /* produce the histogram of the image */
18.     upc_forall(i=0; i<N; i++; i*THREADS/N)
19.     {
20.         for(j=0; j<N; j++)
21.         {
22.             upc_lock(lock);
23.             hist[img[i][j]] ++;
24.             upc_unlock(lock);
25.         }
26.     }

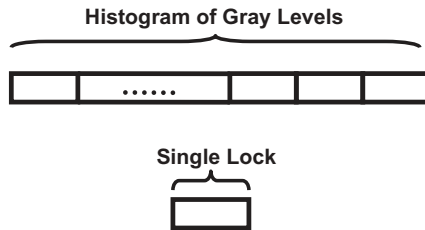
27.     upc_barrier;

28.     /* print out the histogram */
29.     if(MYTHREAD == 0)
30.         print_hist();

31.     return 0;
32. }

```

In a parallel histogramming program, the image data can be distributed by chunks of rows across the threads. In line 3 we do exactly that for a  $1024 \times 1024$  image which is distributed by blocks of size equal to the largest possible chunk of rows. An array of 256 elements can be then be used to hold the histogram, line 4. However, the array of locks can have any size from 1 to 256, controlling accesses to parts of the histogram with the desired granularity. Each thread examines its own pixels and increments the corresponding histogram index every time the corresponding pixel value is found. The problem is, however, that two of the threads may find the same pixel value locally and attempt to increment the same cell at the same time, which creates a race condition. Thus, the result from one of those operations may be lost. Therefore, a lock is needed to treat the histogram-updating block of statements as a critical region. A lock is therefore declared in line 5. Line 6 through 11 are for a function called `initialize`, which allocates a single lock using `upc_all_lock_alloc` and checks whether it has indeed been allocated. The



```

upc_lock( lock );

hist[ img[ i ][ j ] ]++;

upc_unlock( lock );

```

**Figure 6.4** Histogram Algorithm Using Single Lock

main program starts by calling `initialize`, then a barrier to make sure that the lock has been created before the threads start to use it, lines 15 and 16. The `upc_forall`, lines 18 through 26, is where the processing takes place, and each parallel iteration corresponds to processing one image line and updating the histogram accordingly. The `upc_forall` distributes the work across all threads by chunks of iterations such that each thread gets the chunk of iterations that corresponds to processing its chunk of rows of image data. The body of the inner loop is considered a critical section since it is where one pixel of the image is examined and the corresponding histogram entry is incremented. Therefore, the lock is locked at the beginning of the inner loop body and unlocked at the end of it (Figure 6.4).

The histogramming problem implementation of Example 6.3 suffers from a simple but serious problem. There is only one lock and it has to be used by all threads, even if two threads are trying to update different entries in the histogram, in which case these two operations should have been performed in parallel. To resolve this problem and enable exploitation of the maximum possible parallelism, we could use as many locks as the number of entries in the histogram (Figure 6.5). Thus, each histogram entry is controlled individually. This is shown in Example 6.4. Example 6.4 does not give the full code but only highlights the differences between the single- and multiple-lock implementations.

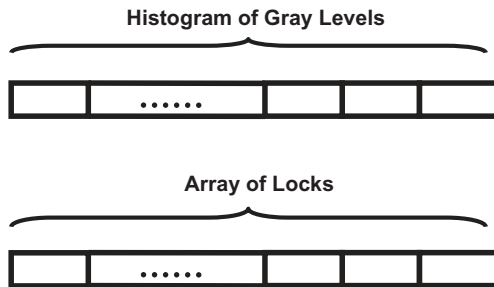
**Example 6.4:** `hist2.upc`

```

...
1. upc_lock_t *shared lock[256];

2. void initialize(void)
3. {
4.     int i;

```



```

upc_lock( lock[img[ i ][ j ] ] );

hist[img[ i ][ j ] ]++;

upc_unlock(lock[img[ i ][ j ] ] );

```

**Figure 6.5** Histogram Algorithm Using an Array of Locks

```

5.  upc_forall(i=0; i<256; i++; i)
6.  {
7.    lock[i] = upc_global_lock_alloc();
8.    CHECK_MEM(lock[i]);
9.  }
10. // initialize the image img[][]
11. }

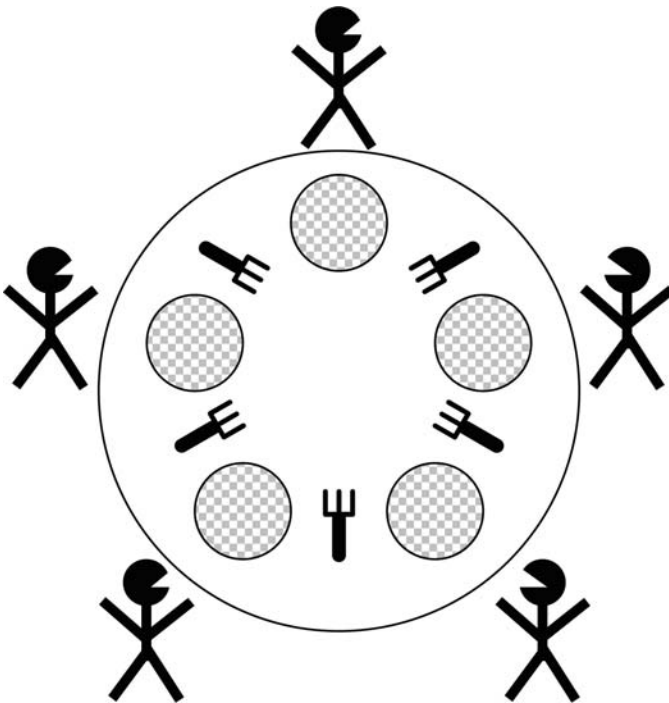
12. int main(void)
13. {
14.   ...
15.   upc_forall(i=0; i<N; i++; i*THREADS/N)
16.   {
17.     for(j=0; j<N; j++)
18.     {
19.       upc_lock(lock[img[i][j]]);
20.       hist[img[i][j]] ++;
21.       upc_unlock(lock[img[i][j]]);
22.     }
23.   }
24.   ...
25. }

```

Line 1 declares a shared array of locks to hold the histogram locks. Note that this array cannot be private since all entries should be accessible from all threads. The `initialize` function, lines 2 through 11, uses the noncollective allocation call `upc_global_lock_alloc` inside a `upc_forall`, which allocates a 256-

element array of locks, `lock[ ]`. In `main()`, each time a thread tries to increment a histogram entry, it uses the corresponding lock.

In Example 6.4 there was no room for deadlocks, a problem that can occur when multiple resources are needed to perform one service and each of the service requesters gets and holds on to part of the resources needed, such that none of the requesters would ever get all needed resources at once. Imagine two friends eating in a Chinese restaurant, and there is only one pair of chopsticks. If each grabs and holds on to one chopstick, they will both starve. As this is a situation that can happen often in system software that manages computer resources, computer scientists abstracted this with the standard computer science problem known as the *dining philosophers* (Figure 6.6). [DIJ71]. In this case a number of philosophers sit at a dining table eating short meals and thinking in between with one fork between every two people. However, each philosopher needs two forks to be able to eat. Each philosopher thinks for a random period of time until he or she gets hungry, then attempts to grab two forks, one from the left and one from the right, to start eating the meal. If the person succeeds, he or she starts eating. If the person does not get any fork, he or she continues to try. If the person gets only one fork, he or she puts it back. This problem can be modeled with one lock per fork. If `upc_lock` is used, then if a philosopher gets one fork only, he or she must



**Figure 6.6** Dining Philosophers Problem

unlock it. This is a possible scheme and the reader is asked to develop this program as an exercise at the end of the chapter. It requires careful programming to ensure that deadlock does not occur. However, `upc_lock_attempt` may simplify things quite a bit in this case.

**Example 6.5:** philosophers.upc

```

1. #include <upc_relaxed.h>

2. typedef enum{THINKING, STARVING, EATING} philosopher_status_t;
3. upc_lock_t *shared a_fork[THREADS];

4. void initialize(void)
5. {
6.     a_fork[MYTHREAD] = upc_global_lock_alloc();
7.     CHECK_MEM(a_fork[MYTHREAD]);
8. }

9. void life_cycle(void)
10. {
11.     philosopher_status_t state;
12.     int num_meals=0, delay_thinking=1, delay_eating=2;
13.     int left, right, got_left, got_right;

14.     left = MYTHREAD;
15.     right = (MYTHREAD+1)%THREADS;
16.     state = THINKING;

17.     while(num_meals < 20) // life of each philo. is 20 meals
18.     {
19.         if(state == THINKING)
20.         {
21.             printf("Philosopher %2d:--- I am thinking\n", MYTHREAD);
22.             sleep(delay_thinking);
23.             printf("Philosopher %2d: --- I finished thinking, now I
                am starving\n", MYTHREAD);
24.             state = STARVING;
25.         }

26.         // tries to lock both forks
27.         got_left = upc_lock_attempt(a_fork[left]);
28.         got_right = upc_lock_attempt(a_fork[right]);

29.         if(got_left && got_right)
30.         { // got both...

```

```

31.         printf("Philosopher %2d: I have both forks --- I start
              eating\n", MYTHREAD);
32.         state = EATING;
33.         sleep(delay_eating);
34.         num_meals++;

35.         printf("Philosopher%2d: I have both forks --- I finished
eating my %d meal", MYTHREAD, num_meals); if (num_meals > 1)
printf("s"); printf("\n");

36.         // release both forks
37.         upc_unlock(a_fork[left]);
38.         upc_unlock(a_fork[right]);

39.         state = THINKING;
40.     }
41.     else
42.     { // either one or none of the fork were available
43.         if (got_left)
44.             upc_unlock(a_fork[left]);
45.         if (got_right)
46.             upc_unlock(a_fork[right]);
47.     }
48. }
49. printf("Philosopher %2d: *** I ate too much, I am leaving!
***\n", MYTHREAD);
50. }

51. int main(void)
52. {
53.     // should be run with >=2 THREADS
54.     initialize();
55.     upc_barrier;

56.     life_cycle();
57.     upc_barrier;

58.     if (MYTHREAD == 0)
59.         printf("*** --> all philosophers left the table <-- ***\n");

60.     return 0;
61. }

```

Example 6.5 shows an implementation of a variant of the dining philosophers problem in which each philosopher is restricted to 20 meals, using the

`upc_lock_attempt`. In this example, line 2 defines an enumerated type to indicate the philosopher status as thinking, starving, or eating. Line 3 declares an array of pointers to locks, one to represent each fork. Note that each philosopher is an independent actor, and therefore each philosopher will be represented by a thread. Lines 4 through 8 are used for initialization by creating the locks, one per thread using the noncollective `upc_global_lock_alloc` with all threads making the call, and checking whether the allocation took place correctly. Lines 9 through 50 comprise the `life_cycle` function, which sequences through a philosopher's activities. Line 11 defines `state` as a private holder of the local philosopher state. In line 12, the number of meals eaten by a philosopher are initialized to zero, and the delay for thinking and eating are set to 1 and 2 seconds, respectively. Lines 14 and 15 are used to identify the two forks sought. Line 15 uses modulo arithmetic to emulate the round dining table. Line 16 initializes all philosophers to the thinking state. According to line 17, each philosopher can eat only 20 meals, after which the program should exit. Such a limit is introduced to terminate the program in a reasonable time. Lines 19 through 25 show how the philosopher will work. If he or she is thinking, the philosopher will sleep for the thinking delay. When the philosopher awakes, his or her state changes to starving, line 24.

In line 27, each philosopher attempts to get the left fork, and if so, the program sets the corresponding flag. The philosopher then tries to get the right fork, and if he or she succeeds, starts eating, which amounts to setting the state to eating and sleeping for the eating delay. The philosopher then increments the number of meals that he or she consumed and prints messages to this effect. The philosopher also releases both forks at that time and switches his or her state to thinking.

Lines 41 through 47 deal with the situation when the philosopher has been able to obtain one fork or no fork. In this case the philosopher releases the fork that he or she got, if any, and stays in the starving state.

If the 20 meals are consumed by a philosopher, he or she leaves the table and the function returns to the `main`. Thus, the `main` function, lines 51 through 61, initializes and waits for all threads to create their locks, gets the life cycle of the philosophers under way, and uses a barrier again to make sure that all philosophers are done before printing out the relevant statistics and exiting.

## 6.4 MEMORY CONSISTENCY

*Memory consistency* refers to the order of memory accesses within one thread and when changes to a shared object by one thread become visible to other threads. In many cases, systems might find opportunities to optimize the performance by exchanging the order of instructions, which causes the reordering of memory references within one thread. In addition, although the programmer may wish to see all changes to shared objects become visible to all threads immediately, this might be associated with performance penalties.

Under UPC, references to shared objects are either strict or relaxed. For *relaxed references*, there is no change to the C Standard execution model as applied to an

individual thread. This implies that systems are free to reorder and/or ignore operations (including shared operations) if it appears that such changes will not affect the final results of the code. For *strict references*, such reordering is not allowed. In addition, strict references cause all prior shared references to become visible to other threads.

UPC provides several mechanisms to control the underlying memory consistency model at various granularities:

1. The shared object level
2. The block statement level
3. The program level

At the shared object level, UPC allows the use of `relaxed` or `strict` as qualifiers for **shared** declarations. For example, one can declare

```
strict shared int x;
```

where `strict` is a reference-type qualifier. For controlling accesses in a block of statements, one can use pragmas: for example,

```
{
  #pragma upc strict
  ...
}
```

When a `pragma` appears outside a compound statement, it applies until the next `pragma` or the end of the program or until another such `pragma` is encountered. If a `pragma` appears inside a compound statement, it only affects the code that follows until the end of the compound statement.

Finally, the memory consistency model can be controlled at the program level by including one of two possible files:

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

or

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

When more than one of the mechanisms described previously are used, the reference-type qualifier has the highest precedence, followed by pragmas, followed by the `upc_strict.h` or `upc_relaxed.h` file. Note that in the absence of such reference qualifiers, pragmas, and `upc_strict.h` or `upc_relaxed.h` file, the behavior follows the default instituted by the implementation. Another way to make modifications to shared objects known to all threads is by using the



fence construct, `upc_fence`. This forces all previous shared references to complete. Thus, the `upc_fence` statement is equivalent to a *null* strict reference. This ensures that all shared references issued before the fence are completed before any following it are issued. It also ensures that no shared references appearing after the fence are executed before it.

One example to demonstrate some of the concepts that have been just described can be derived from the computer science problem known as the *bakery algorithm* [LAM74]. In this problem customers arrive at a bakery store. As a customer arrives he or she gets a number. Multiple customers can arrive at the same time, and therefore these numbers may not be unique. In that case, the order of service is resolved in some other way, such as age. The problem, however, is that the bakery does not provide a way (e.g., a display) for the customers to learn of their turn. Instead, the customers have to communicate and determine who has the priority to be the next to receive service. Memory consistency issues in this example arise in three cases. First, when a customer arrives, the person should let the others know that he or she is going to be picking a number. When the person gets a number, the number chosen changes what the next available number will be. Finally, when the person gets a ticket, he or she should let the others know. In each of these cases, all threads must become aware of the modified data that reflects these facts as soon as the change occurs.

**Example 6.6:** `bakery1.upc`

```

1. #include <stdio.h>
2. #include <upc_relaxed.h>

3. typedef enum {FALSE, TRUE} bool_t;
4. strict shared bool_t choosing [THREADS];
5. strict shared int number [THREADS];
6. strict shared int counter;

7. int main(void)
8. {
9.     int i;

10.    if (MYTHREAD == 0)
11.        counter = 0;

12.    choosing[MYTHREAD] = TRUE;
13.    upc_barrier;

14.    number[MYTHREAD] = (++counter); // pick up a ticket number
15.    choosing[MYTHREAD] = FALSE; // announce that I got my ticket

16.    for(i=0; i<THREADS; i++) //check all other customer
17.        {

```

```

18.         while(choosing[i] == TRUE); // wait for customer i
           // to get his/her ticket number
           // is Customer i still in shop AND will he be served before me?
19.         if((number[i] > 0) &&
20.           ((number[MYTHREAD] > number[i]) ||
21.            ((number[MYTHREAD] == number[i]) && (MYTHREAD>i))))
22.           while(number[i] > 0); // so I need to wait for him/her
           // to leave the shop before
           // considering been served
23.     }
           // I got my turn, I have the lowest ticket or I am the youngest
           // (smallest thread number) among my peers
24.     printf("Thread %02d exits with number=%d\n", MYTHREAD,
           number[MYTHREAD]);
25.     number[MYTHREAD] = 0; // I throw my tix ... that's done for me
26.     return 0;
27. }

```

In line 2, the general memory consistency model is set to `relaxed` at the whole program level. Line 4 is creating a shared array of boolean type to reflect whether a particular customer (i.e., thread) is now choosing a number. It is reference qualified to be strict, to ensure that when a thread is choosing, all others will know that immediately. Similarly, the number that is chosen by a customer is also declared (line 5) in a somewhat similar manner, so that when a thread picks up a number, the rest will know of it immediately. The `counter` has a similar declaration, although it is a scalar since it only indicates the current available number to be picked by the next customer, line 6.

Lines 10 and 11 initialize the process by setting the `counter` to 0. Then line 12 gets all threads to start choosing by setting their `choosing` flag to `TRUE`. A barrier in line 13 guarantees that all customers start competing for the numbers simultaneously. Thus, each of them simultaneously tries to preincrement and get the value of counter to be its own number, which would be a minimum of 1, line 14. This means that some race conditions might occur. This is not a problem since, at worst, that can give more than one thread the same number, and as we said, that will be resolved in a different way. Then each of the threads sets its `choosing` flag to `FALSE`, lines 15. Lines 16 through 23 provide a loop that goes through the threads one by one. For each thread, if the number has been chosen, the thread starts figuring out whether it now has the priority to receive the service. In line 16, a loop examines the threads one by one, skipping over those who are not currently choosing, line 18. Lines 19 examines if the current number is over a zero, which means that the customer is still in the bakery. Line 20 checks to see if the number of the executing thread is greater than the numbers of the other threads, which means that the current thread does not have the priority to receive service yet. Line 21 checks whether the executing thread has a number similar to that of any of the other threads. This constitutes a conflict. The conflict is eliminated by selecting the thread that has the lower thread number, or the youngest customer.

**Example 6.7:** bakery2.upc

```

1. ...
2. shared bool_t choosing[THREADS];
3. shared int number[THREADS];
4. shared int counter;

5. int main(void)
6. {
7.     int i;
8.     ...
9.     upc_barrier;

10.  {
11.     #pragma upc strict

12.     number[MYTHREAD] = (++counter);
13.     choosing[MYTHREAD] = FALSE;
14.  }

15.  for(i=0; i<THREADS; i++)
16.  {
17.      while(choosing[i] == TRUE);

18.      if((number[i] > 0) &&
19.         ((number[MYTHREAD] > number[i]) ||
20.          (number[MYTHREAD] == number[i] && (MYTHREAD>i))))
21.          while(number[i] > 0);
22.  }

23.  printf("Thread %02d exits with number=%d\n", MYTHREAD,
number[MYTHREAD]);
24.  {
25.      #pragma upc strict
26.      number[MYTHREAD] = 0;
27.  }

28.  return 0;
29.  }

```

Example 6.7 uses **pragma** to create the same effect. The **pragma** in line 11 is inside a compound statement and will apply to the rest of the statements in that block, lines 12 and 13. Similarly, the **pragma** in line 25 applies only to the statement in line 26. These were all intended to ensure that updates to the shared objects `number`, `counter`, and `choosing` become immediately visible to all

threads. Example 6.8 provides a variant implementation of the bakery algorithm that accomplishes the same results using `upc_fence`.

**Example 6.8:** bakery3.upc

```

1. ...
2. shared bool_t choosing[THREADS];
3. shared int number[THREADS];
4. shared int counter;

5. int main(void)
6. {
7.     ...
8.     upc_barrier;

9.     number[MYTHREAD] = (++counter);
10.    choosing[MYTHREAD] = FALSE;
11.    upc_fence;
12.    for(i=0; i<THREADS; i++)
13.        {
14.            while(choosing[i] == TRUE);

15.            if((number[i] > 0) &&
16.                ((number[MYTHREAD] > number[i]) ||
17.                 ((number[MYTHREAD] == number[i]) && (MYTHREAD>i))))
18.                while(number[i] > 0);
19.        }

20.    printf("Thread %02d exits with number=%d\n", MYTHREAD,
21.           number[MYTHREAD]);
21.    number[MYTHREAD] = 0;
22.    upc_fence;

23.    return 0;
24. }
```

## 6.5 SUMMARY

UPC is a parallel processing language that requires coordination among the executing threads. Therefore, UPC provides for synchronization and memory consistency control. Examples of these mechanisms are barriers, locks, and fences. UPC barriers can ensure that all threads reach a given point before any of them can proceed further. Locks are needed to coordinate accesses to critical sections of the code. Finally, memory consistency control gives the ability to control the access

ordering to memory by the various threads such that performance can be maximized with no risk of data inconsistency. Memory consistency mechanisms in UPC also give the programmer the ability to decide when changes to a shared object become visible to other threads.

## EXERCISES

- 6.1 Implement a program calculating the  $n$ th power of a square  $N \times N$  matrix (using `upc_barrier`).
- 6.2 Develop a version of the philosophers problem using regular locks [`upc_lock()` and `upc_unlock()`, no `upc_lock_attempt()`] and shared variables as semaphores if needed.
- 6.3 Implement a variant of the heat conduction problem that uses `upc_locks` for the all-reduce operation at each iteration.
- 6.4 Implement a queue data structure in UPC (using `upc_lock_t`).
- 6.5 Develop the sleeping barber algorithm using the queue from Exercise 6.4. The sleeping barber algorithm works as follows: At the beginning of the day, the barber sleeps and will wake up as soon as a customer enters his shop. There are  $n$  seats in the shop. A total of  $N$  customers are entering his shop randomly all day long to have their hair cut. If a seat (among the  $n$ ) is free, the customer sits and waits (if the barber is busy) or the barber will be serving this customer immediately. If all seats are full, the customer will be back after a random amount of time. The barber finishes his day after all  $N$  customers are served.

# Performance Tuning and Optimization

The UPC language has been designed specifically to allow programmers to get the best performance from a wide range of parallel computer architectures. Many features of the UPC language and toolset will aid in this task. But to obtain the best performance, it is essential to have in mind some basic knowledge about the characteristics, system architecture, and performance of parallel computers. The chapter begins with a brief primer on parallel machines and general performance issues in parallel programming. For a deeper insight into many of these issues, there are many texts on the subjects referred to in this chapter. After this general introductory overview of parallel system architecture, three critical factors in achieving performance with UPC are discussed:

1. The UPC compiler, which analyzes the UPC application program and applies a variety of techniques to it to produce good executable code
2. The UPC run-time system, which both enables running programs and observes their dynamic behavior in order to undertake actions to improve performance at run time
3. Hand optimizations that are performed by programmers to enhance application performance

With this arsenal of language constructs and software tools, the programmer has the means to craft parallel programs and make them perform well. The UPC programming model exhibits significant control flexibility to permit some important optimizations. Some specific techniques for achieving these are discussed in this chapter. In particular, methods for taking advantage of data locality, improved synchronization, and latency hiding are presented. All of these useful practices are illustrated through detailed case studies.

## 7.1 PARALLEL SYSTEM ARCHITECTURES

UPC provides a semantic bridge between the user application problem and the target physical parallel system architecture, through the software medium of compilers and a run-time system. Although the semantic correctness should be a constant among separate machines of different scale and class, their performance properties may differ, sometimes dramatically. Several classes of parallel systems are in common use today. At least to some degree, each may vary in scale, perhaps by several orders of magnitude (measured in the number of computing elements). Although this has not always been the case, the prevalent parallel systems all employ a plethora of processors, memory banks, and internal communications networks [system area network (SAN)]. But the different classes of systems differ in the architecture of the microprocessors, the structure or organization of these combined elements, and their logical interrelationships. In this section we describe briefly the dominant classes of parallel system architectures, to provide a foundation with which to consider driving performance properties and optimization methods to address them.

Five classes of parallel system architecture are described. Although not an exhaustive set of the entire domain of parallel computing architecture, they represent the vast majority of commercial systems and all of the systems to which UPC compilers have been targeted:

1. Symmetric multiprocessors (SMPs)
2. Distributed shared memory multiprocessors (DSMs)
3. Distributed memory multiprocessors
4. Commodity clusters
5. Parallel vector processors (PVPs)

It should be noted that these terms are not defined rigorously, and workers in the field may employ them with slight variation of intent. The following is one reasonable and representative description, but by no means the only credible one.

For the sake of simplicity, the following parallel system descriptions will emphasize two architecture properties: degree of coupling, and memory model. Degree of coupling is generally characterized in the range loosely to tightly coupled communication among the separate computing and memory elements comprising the parallel system. But this is a simplification of a combination of the total rate at which data can be moved across the system at one time (its bisection bandwidth), the rate at which any one processing node can inject or receive data from the SAN (its local bandwidth), the time it takes to move the data between any two such nodes (latency), and the possible variation of the latency among different node pairings (UMA versus NUMA), as well as the overhead or work required in the critical execution time to sending and receiving data messages from remote nodes.

The memory model of a system determines the visibility of the distributed main memory from the perspective of the processor node hardware. The three possible conditions that discriminate among system classes are:

1. Full shared memory with cache coherence
2. Shared memory without cache coherence
3. Separate local memory partitions

The combination of these memory models and the degree of communication coupling is sufficient to describe and differentiate the four dominant parallel system architecture classes.

*Symmetric multiprocessors* (SMPs), also known as *uniform memory access architectures* (UMAs), represent a parallel computer architecture in which processors share the same memory physically such that the access time from any processor to the memory system is the same. In such a system, the set of processors have direct and equal access to the set of memory banks. The SAN connects the memories and the processors such that the processors have the same physical and logical access characteristics of true shared memory. Furthermore, coherency across processor caches is maintained consistent so that changes to the state of a variable by one processor are reflected by the copies of that variable in the caches of any of the other system processors. This permits a true shared memory execution model and provides very low overhead for all memory accesses across the system. Nonetheless, there are two important aspects of SMP systems that impose performance bottlenecks. The first is *restricted scale*. To support UMA across the memory system and full cache coherence, hardware mechanisms are employed that do not scale indefinitely. Therefore, the majority of SMPs comprise a small number of processors, two to four being typical and few systems exceeding 16 processors. The second is *restricted bandwidth* to main memory. Because of the need for cache coherence, all memory traffic has to be monitored, and the amount of memory bandwidth available to the processors is limited. When all processors attempt to engage the main memory simultaneously, there occurs significant contention in many cases, degrading overall system performance and efficiency.

*Distributed shared memory machines* (DSMs), also known as *nonuniform memory access architecture* (NUMA), comprise a system in which the memory is physically distributed across the processors but is globally addressable and accessible by all processors. This provides a single machine image. However, a processor can access its local memory modules faster than remote ones, although all of them are shared by all processors. The basic building block in a DSM machine is a node, which can be as simple as one processor with some memory attached to it, or an entire SMP system. The blocks are interconnected together with some interconnection network.

Unlike SMPs, distributed shared memory machines incorporate cache coherence mechanisms that scale. DSM systems with thousands of processors have been deployed commercially. The logical execution model of the DSM is the same as



that of the SMP, including cache coherence across processors. But the performance model is different. There is a partition of memory that is relatively local to a given processor node while the remaining memory is relatively remote. Access to local memory is much faster (shorter latency and higher bandwidth) than accesses to remote memory. The difference can be as much as one to two orders of magnitude for very large systems. A local node may incorporate more than one processor and more than one memory bank in an SMP configuration where all processors within a single SMP node have equal access to local memory. The DSM class of system shares with the SMP system the low overhead of memory access and cache coherence, but suffers from nonuniform memory access times even as it benefits from potentially larger system scale.

Now things get a bit complicated. There is a variation of the DSM system that does not have cache coherence, but there is no agreed term for this class of system. Therefore, we will refer to it here as DSMwoCC, but this is not common usage. This is an important class of systems. It has better efficiency (lower overhead) of memory access than that exhibited by ordinary DSMs because it does not incur the additional hardware overhead times required for the cache coherence. It is also less expensive to build and can scale to larger systems. But DSMwoCC systems also suffer from NUMA behavior and do not guarantee a pure shared memory model such that the software has to take into consideration this potential conflict of copies. DSMwoCC is the closest hardware system to the abstract programming and execution model of UPC and is an ideal target platform for UPC compilers. However, UPC can be mapped as efficiently as other paradigms to other types of architectures.

*Distributed memory (DM) systems*, also known as *multicomputers*, are tightly coupled ensembles of independent processing nodes. Each node comprises one or more processors in an SMP configuration. The nodes are interconnected with a high-bandwidth low-latency custom SAN. This network is used to send messages between the nodes. Logically, a node processor can only directly address memory local to that node. No direct remote memory accesses are possible. This is referred to as a *partitioned or fragmented memory model*. For nodes in a DM system to cooperate on a single computation, explicit message passing between processes on separate nodes must be accomplished. Often, large blocks of contiguous data may be sent in a single message between processes running on separate nodes, amortizing the overhead and latency of communications while exploiting the relatively high bandwidth of the SAN. However, not all applications can benefit from that, as some may be latency sensitive. Although the overheads incur software action at each end and are thus aggravated compared to DSM systems, there are many problems whose data access patterns are such that this additional burden is acceptable. DM systems are less expensive than DSM systems and they can scale to much greater sizes. Systems on the order of 10,000 processors have been deployed, and DM systems of tens of thousands of processors are likely to emerge in the near future. Although the hardware logical model does not directly match the UPC execution model, the UPC compiler and run-time system enable application codes written in UPC to perform correctly on DM systems. However, the performance

model due to the distributed memory semantics of the machine is different from that of the DSM, and performance tuning and optimization requires that such operational distinctions be taken into consideration.

*Commodity clusters* are collections of stand-alone computer systems manufactured for mainstream data processing markets and integrated using COTS local area or system area networks. Clusters of personal computers often running Linux are known as *Beowulf cluster systems* [STE01], named after the early NASA project that introduced them. Due to the benefits of economy of scale of production, commodity clusters deliver exceptional performance to cost for many (but not all applications), exceeding that of DSMs, DMs, and vector machines (see below) by as much as an order of magnitude or more. Today, more than half of the world's top 500 computing systems (measured by the Linpack benchmark) are commodity clusters.

Commodity clusters present the same logical model as that of distributed memory systems. Nodes of one or more processors in an SMP configuration communicate by a message-passing protocol between processes running on the separate nodes. The processors of each node can directly access (using load and store operations) only the memory local to that node. But, if anything, the performance model of a cluster is worse than that of a DM class machine. The overheads for remote communication are larger and the bandwidth and latencies of the COTS network are worse than the custom communication network of the DM systems. Latencies can be one to two orders of magnitude longer and bandwidth an order of magnitude lower than DM machines. Acquiring a remote value can require 1000 to 10,000 processor cycles. Performance tuning and optimization is driven by the need to exploit locality and minimize communication, especially fine-grained communication, even at the expense of performing more work or redundant work on each node. Despite these increased challenges, commodity clusters are likely to be a common target platform of UPC applications for some time to come.

*Parallel vector processors* (PVPs) are typically designed around a proprietary processor that is specially crafted to target vector computations. Vector processors can have vector registers that can hold a complete array; their functional units are designed as pipelines that can exploit the overlapped parallelism in processing the elements of a long array. The instruction sets of vector processors are such that an entire loop to fetch, store, or process an array can be replaced by a single vector instruction. Vector processors can sustain very high bandwidths through their networks to memory for certain types of access patterns, including unity stride and uniform stride accesses. Advanced systems can do sparse accesses efficiently as well with special gather scatter hardware. PVPs can be used in all of the system configurations described above, replacing the commodity microprocessors with custom vector processors. When used effectively, they can provide among the highest delivered performance of any parallel system for certain classes of applications. But the performance tuning and optimization using UPC presents a different set of code structuring trade-offs than other system types described.

UPC does not require any particular computer architecture to run. In fact, there are currently UPC compiler implementations for DSMs, multicomputers, clusters,

and vector machines. However, it should be noted that UPC requires a shared memory view. Thus, implementations on DSMs and SMPs are relatively easier and, depending on the implementation, may be more efficient. The shared space view under UPC requires that multicomputer implementations have a layer that can present the underlying architecture to UPC as if it were a shared memory architecture. Among the software libraries that did this successfully are PGAS [HU00] and Elan [QUA03]. The fact that the shared space under UPC is partitioned makes it map more efficiently on multicomputers than strictly shared space programming paradigms.

## 7.2 PERFORMANCE ISSUES IN PARALLEL PROGRAMMING

### Parallel Overhead

While increasing the number of processors of a single machine increases its overall peak computational power, it also adds a number of overheads and latencies that can degrade the potential benefit of parallel processing. Parallel programmers have to consider these overheads, latencies, and other system factors that reduce performance to enhance scalability and execution efficiency. Scaling here means getting more measured performance as the number of processors increases. The main types of potential overheads and latencies that inhibit performance are:

- Interprocessor communications
- Processor synchronizations
- Load imbalance
- Redundancy

*Interprocessor communications* are required so that the many processors of a parallel machine can work together effectively on the same computational problem. At the beginning of a computation, initial input data may need to be distributed across the processor nodes. During the computation, processors may need to exchange intermediate results. By the end of the computation, the final result data will need to be integrated to be directed ultimately to external outputs. In the uniprocessor case, none of these operations are required. In the parallel case, they are necessary but they take the processor away from its main purpose, which is computing. Overhead from both controlling the communications and latency of moving the data between processor contributes to the potential performance degradation. Therefore, interprocessor communications must be kept to a minimum.

*Synchronization* deals with the need to order and coordinate the work of processors (threads) to ensure that results obtained from parallel computations are similar to that we can obtain from a uniprocessor machine. We covered the topic of synchronization in Chapter 6. One thing that should be noted is that synchronization comes at a nontrivial cost to performance, and since it is not part of the actual computing, it is another overhead. Programmers interested in maximum performance

should certainly reduce the amount of process and data synchronization and use the minimum to guarantee the correctness of the results.

*Load imbalance* is another source of inefficiency. It is very difficult in many cases to divide the data and work equally among all processors. Sometimes the size of the data is not divisible by the number of the processors, and sometimes some data require more processing than others, as in the case of zero and nonzero elements of a sparse matrix. Also, sometimes the workload changes dynamically at run time. In these cases some of the processors will complete their work much later than the rest, which delays the processing time of the overall machine. Programmers should strive to balance the work across the processors, but this may not always be possible.

*Redundancy* refers to the fact that in many cases exactly the same computations take place on all processors. For example, consider the problem of initializing a particular variable that is needed at all processors. One option is to initialize it at only one processor, then broadcast it. The other is to initialize it at all processors. In the former case we suffer communications overhead. In the latter we perform an operation that we used to do once in a sequential machine many times, which is a redundancy overhead that was necessary for the parallel execution to take place.

## Machine Imbalances

One problem that aggravates communication overhead is the unbalanced rate of improvements in processor and interprocessor communication technologies. Processor performance has been doubling every 18 to 24 months, but speed of communication is not growing at the same fast pace. The same problem also occurs from the memory perspective, where processing speed is growing at a much higher rate than improvements in memory latency and bandwidth.

The performance of an interconnection network can be analyzed in many ways. However, two important metrics are bisection bandwidth and latency. Bandwidth measures the throughput of the machine; *bisection bandwidth* refers to the amount of data that can be transferred between both halves of a machine after we split it from the point of the weakest bandwidth. *Latency* is the amount of time taken to send the minimum transferable size of data from a source node to a destination node. Network traffic is generally pipelined. This means that the first data packet sees the full latency, but the following ones will be arriving in sequence immediately after the first packet is received. Programmers should develop an understanding of how to minimize communications overhead and/or its impact.

There are three ways to minimize the impact of communication overhead: locality exploitation, amortization, and overlapping. *Locality exploitation* refers to attempting to complete all work associated with a part of the data, before moving to the next. Locality exploitation can reduce the amount of data that needs to be accessed remotely. Locality exploitation also optimizes the operation of the memory hierarchy by increasing the likelihood of finding what we need in the cache.

Since the first packet sees the full latency delay, the programmer should try to make the messages as large as possible. This *amortizes* the initial overhead over

the largest possible number of packets. However, such optimization is required only on architectures where communication networks may be suffering from high latency and no hardware cache support over shared memory spaces. There are two ways to do this: one is by integrating all of the data that needs to be communicated, even if they are unrelated, to produce a smaller number of messages. The other strategy is to increase the computational granularity. This can be accomplished by distributing data by the largest possible chunks and attempting to complete the work on that chunk locally prior to any exchange of data. In many applications, aggregating and maximizing the size of communication messages may not be possible. In these cases, UPC continues to have a great advantage in that it can perform short transfers efficiently, due to its shared memory view.

In addition, when possible, programmers can *overlap* communications with computation. Instead of in-demand synchronous communications, programmers can start reading/receiving data asynchronously well ahead of time, then ensuring completion of the asynchronous operation only before the relevant data is needed for processing.

### 7.3 ROLE OF COMPILERS AND RUN-TIME SYSTEMS

Getting a program to run on a computer involves a number of tools and processes that need to be understood, to some degree, to effectively optimize the performance of programs. Of course, it all starts with algorithm development and coding in UPC source code. A compiler is then used to translate the UPC source code to machine code for a specific computer. Next, a run-time system is responsible for loading the machine code into the system and starting execution. Since UPC is a parallel language, the run time is also parallel and provides a number of services to the executing program, such as I/O and parallel synchronization. Finally, some tools help analyze both the correctness of programs and their performance. These components interact in interesting ways that should be understood before effective optimization can be performed.

#### Compilers

UPC compilers are becoming widely available. Each one is produced by a company or other organization to support the preparation of UPC programs on one or more computer systems. A “correct” UPC compiler must take all “valid” UPC programs and ensure that the program, when executed, performs according to the rules of UPC. Appendix A provides the most recent specification of what is “valid” in terms of both the expression of UPC programs and their execution. Although most readers of this book will not need to study the specification, it is the final arbiter of what a UPC program is.

The compiler always has a second goal beyond the simple correct execution of a program: namely, the efficient execution of the program. This is possible because the UPC specification allows compilers to change UPC code into, potentially, many

different sequences of machine instructions as long as it appears as though the program produces a “valid” execution. How does a compiler decide which sequence is both valid and will yield the most efficient use of the computer? Although the answer to this has been the subject of about 40 years of compiler research, some things are clear. First, the compiler must understand which sequences are efficient on the machine. Second, the compiler must be able to “see” all the activity that affects performance. For example, compilers generally have a hard time telling what a program is doing in its subroutines, especially if they are in a library. Of course, compilers will have diverse “skills” at this task, depending on the abilities of those who developed the compiler, how much effort they put into its development, and the quality of the computer that will execute the program.

## Runtime

The run-time system works with the machine code produced by the compiler and is responsible for the correct and efficient execution of UPC programs. It is usually created and supported by the group that produced the compiler. The first thing the run-time system does is to load the program into the computer memory and cause its execution to begin. This means that for each thread, the run time will cause the programs `main()` routine to be called, then requests the program to perform specific basic functions. For example, run-time support is often needed to implement the `upc_notify` and `upc_wait` statements. The run-time system is usually responsible for such duties as interacting with the operating system when performing I/O functions and supporting debugging and performance analysis.

To achieve efficiency, the run-time system will often be able to observe certain characteristics of the program execution in real time and respond by taking anticipatory actions. For example, in the UPC programming model, there are many cases in which relaxed shared memory references may be prefetched, cached, deferred, or grouped. On some architectures such optimizations will have a dramatic effect on the performance of applications. Another characteristic of the run time to consider are the relative efficiency of shared memory accesses, locks, and barriers, so that one tailor the synchronization mechanisms chosen to achieve the best performance possible. Many UPC implementations have similar performance ratios for these mechanisms, so techniques learned on one implementation are usually relevant to others.

## 7.4 UPC HAND OPTIMIZATION

Hand optimization is a process by which a programmer attempts to “help” the compiler and/or run-time system to achieve more efficient execution of the program. In case of a new language, such as UPC, programmers might have to employ more hand optimizations until compilers mature enough to incorporate all possible optimizations and offer them to the user automatically, or through simple compile-time mechanisms such as flags. Hand optimizations can be divided into two types:

general optimizations that apply to all systems, and machine-specific optimizations that apply only to a single (or limited class) of systems. Because system structures, mechanisms, and timing characteristics differ, sometimes substantially, among machine types, optimizations of parallel codes applied for one system may have deleterious effects on the performance of another system for which the specific optimizations were not targeted. Care is essential in this respect. Apparently innocuous modifications to the program may invoke subtle changes in system operation that yield dramatic consequences in performance, either positive or negative. Finally, it should be noted that hand optimizations often tend to obscure the readability of code, so it may well be harder to reuse or debug.

There are two main areas to consider for general hand optimization: helping the system (both compiler and runtime) to determine what is “local” to a thread, and helping the system to determine dependences between both private and shared accesses. As will be seen in the following sections, the determination of what is local is best performed by using the UPC affinity concept. There are many techniques for helping the system understand the true dependencies in a program, but they are all required because the rules of the language (UPC in this case) are stricter than the rules of the algorithm. For example, the compiler must perform the code

```
for (i=0; i<n; i++)
    x[y[ i ] ] += 17;
```

correctly even in the presence of the pathological case where all the elements of index array `y[ ]` contains the same value. To execute the code correctly, it must complete the previous store of `x[ ]` before beginning the fetch of the next. This can obviously lead to poor performance. However, if the user wrote the code as

```
for (i=0; i<n; i+=2)
{
    t1 = x[y[ i ]];
    t2 = x[y[ i+1 ]];
    x[y[ i ] ] = t1 + 17;
    x[y[ i+1 ] ] = t2 + 17;
}
```

the compiler is free to fetch two values of `x[ ]` at the same time, because the user has implied that even/odd values of `y[ ]` are different. This would result in better performance on most systems. Of course, this implication could be false and the user might discover a nasty bug. This illustrates that in performance-critical sections of a program, it might pay to write code in a more complex fashion so that the compiler can understand the actual requirements of the algorithm.

The UPC programming model and language provide a plethora of mechanisms for enabling optimization of parallel programs. In this section we present a number of these mechanisms along with an explanation of why they are effective. We begin

with an in-depth discussion of data locality exploitation, and optimization, followed by the effective use of locks and barriers, with special emphasis on the split-phase barriers that UPC provides. Finally, we discuss the importance of using bulk data transfers, concentrating on the difference between UPC bulk transfers and the message-passing model's use of bulk data transfers.

## Memory Locality and Consistency

As discussed above, one of the critical optimizations in programming modern parallel machines is to reduce the number of accesses to memory which are not “close” to the thread issuing the access. UPC's memory model is ideal for controlling such accesses: They may be reduced or eliminated by forming algorithms that access mainly local data or by strategically copying shared data to the private space where accesses are much more efficient. Figure 2.2 depicts the fundamental UPC memory model introduced in Chapter 2. It shows that each thread has access to two data areas: private and shared. The great thing about the private area is that it will be guaranteed to provide efficient access on any UPC system. Shared data provides the means of saving space by sharing data structures among the threads and enabling complex algorithm interactions.

Another feature of the UPC programming model that is critical to performance enhancement and tuning is the concept of affinity. This concept allows each thread to consider a well-defined portion of shared data that is local to the thread and therefore provides the same performance benefits as its private data. This gives the UPC programmer the ability to share data globally but operate on it locally, both contributing to efficient execution. The main mechanisms for taking advantage of affinity are the “casting” of pointers to shared data into pointers to private data and use of the `upc_forall()` construct. More on these points follows in the next sections.

The memory consistency model of UPC can be an important aid in achieving good performance on many parallel systems. This model, in which references are marked as either `strict` or `relaxed`, allows compilers and run-time systems to deliver a combination of good performance when that is needed and tight consistency semantics when required logically. The key here is that these concepts relate to the order in which other threads see a given thread's shared memory accesses. In general, compilers and run-time systems will operate at higher efficiency when they are allowed more flexibility in the ordering of accesses. To understand this range, let's look at the extreme positions of all `strict` and `relaxed` shared references in a program, from both a performance and a consistency point of view.

In the `strict` case, every thread is guaranteed to observe the results of shared accesses in the order in which they are issued. Generally, this is important only when one thread makes a decision based on a value in the shared space: for example, code such as

```
strict shared int x;
strict shared int y;
```



```

thread 1:          thread 2:
x=0;                while (y == 0)
y=0;                /* do nothing */;
x=1;                printf("%d", x);
y=2;

```

A logical analysis of this program indicates that the only value that could possibly be printed for `x` would be 1, since `y` was set to 2 after `x` was set to 1 in thread 1. The only way to guarantee that this is the case is to declare `x` and `y` `strict`. To do this, the implementation of thread 1 must ensure that the value of `x` be updated before `y` is set. This induces a performance implication if it takes time, which on many systems it does. Thus, the all-strict case will make analysis of the type of code written above easier and have a very well defined behavior, but that comes at a cost to the efficient execution of the program.

Now consider the all-relaxed case. The only guarantees are that at synchronization events (e.g., `upc_notify`, `upc_wait`, and `upc_fence`) the side effects of all previous shared accesses have been completed. Consider the following code:

```

relaxed shared int x;
relaxed shared int y;

thread 1:          thread 2:
x=0;                while (y == 0)
y=0;                /* do nothing */;
x=1;                printf("%d", x);
upc_fence;
y=2;

```

The code will again ensure that the result value printed out is 1. But the first accesses may be performed in any order. In fact, the first store of 0 in `x` does not even have to happen, as it has no effect on the program's execution. Obviously, this code will perform better. In general, it is a good idea to use relaxed references for performance gain except when the execution of some other thread depends on the order of accesses seen, as was just shown.

## Locks and Barriers

Beyond the parallel program coordination role played by shared memory as described in the earlier section on shared memory consistency, UPC provides two native coordination mechanisms: locks and barriers (introduced in Chapter 5). A lock is an object in shared space that allows a thread to perform operations called `upc_lock()` and `upc_unlock()` to guarantee mutual exclusion with respect to the lock. A barrier is a collective operation which ensures that all threads have completed all operations prior to the barrier operation before any threads continue

beyond the barrier operation. The major difficulty with barriers and locks is that they can lead to very inefficient operation if not applied carefully.

## Bulk Transfers and Collectives

The UPC library provides a variety of mechanisms that allow the efficient transfer of data between various combinations of shared and private space and support a rich set of collective operations. The principal benefit of these libraries, all of which can be written as UPC programs, is that they offer semantics that are both useful to programmers and may be implemented quite efficiently on a variety of UPC platforms.

The bulk data movement operations [`upc_memget()`, `upc_memput()`, and `upc_memcpy()`] allow efficient movement of data. In most cases, implementations operate near the hardware bandwidth of the underlying communication network. The `upc_memget()` function is used to transfer data from shared space to private space, `upc_memput()` transfers data from private space to shared space, and `upc_memcpy()` transfers data from one shared space to another. In the next section we show examples of using both `upc_memget()` and `upc_memput()`.

## Optimizing for Locality and Load Balance

We now turn to some more concrete examples of using UPC to achieve efficient execution. These are examples of a set of specific optimizations that are generally applicable to many of today's parallel machines. First to be addressed are techniques to enhance performance by exploiting locality. Then the use of split-phase barriers is addressed in some detail. The section concludes with a discussion of the use of locks.

The most elementary optimization for locality is to make a local copy of a remote shared data item that will be reused often. Consider the following transformation:

<pre>shared int x; <u>original:</u> for (i=1; i&lt;big; i++)   f(x);</pre>	<pre><u>optimized:</u>   int local_x;   local_x = x;   for (i=1; i&lt;big; i++)     f(local_x);</pre>
--	---

If the compiler could look inside the function `f()` (and all of the functions it calls) and see that `x` is never changed, it could do the optimization itself. Unfortunately, compilers often cannot perform this sort of analysis: for example, if the function in question is in a library. The benefit of the optimized version is that it reduces the global number of fetches of shared references. The nonoptimized version fetches `big` times, while the optimized version fetches `x` only once, avoiding `big-1` shared references.

The `upc_memget()` library function is perfect for making local copies of larger data structures efficiently, so it might be good to use that if `x` were an array, for example. By writing the code like this, the programmer is asserting that `f()` does not change `x`. If it did, we would have very bizarre behavior. Another potential problem with copies is that they take some time, and if only a small portion is accessed, it might not yield actual performance improvements. Of course, it will also require more memory for the copies on every thread.

If we are unable to make copies of data (either because it might be updated or there is no room for the copy), another approach is to use the affinity concept in UPC. Recall from earlier chapters that each elementary object in the shared space has affinity to a single thread, which allows us to do local computations on shared data. Consider the following transformation:

```
shared struct xs *xp;
original:
struct xs *mine;
for (i=0; i<big; i++)
    if (i%THREADS == MYTHREAD)
        f(&xp[i]);

optimized:
struct xs *mine;
for (i=0; i<big; i++)
    if (upc_affinityof(&xp[i]) == MYTHREAD)
    {
        mine = (struct xs *) &xp[i];
        f(mine);
    }
```

Here we have instituted an “owner-computes” method in which the thread with affinity to the data structure will actually compute the function `f()`, and in the optimized case, `f()` is actually called with a pointer to the shared data that appears to be pointing to private. This will cause `f()` to be an entirely local access function that will be compiled very efficiently. This loop can easily be rewritten using the `upc_forall()` construct. In general, using owner-computes and casting pointers to shared data to private is a foolproof way of getting better performance. Care is required, though: Casting a pointer to shared, which does not have affinity to the current thread, will usually produce an undefined (and usually bad) behavior in your program!

## 7.5 CASE STUDIES

The following case studies are examples of real UPC codes that demonstrate a number of very useful techniques of hand optimization for extracting

enhanced computing efficiency and performance based on earlier discussions in this chapter.

## Space Privatization

Example 7.1 demonstrates how to apply space privatization to our heat conduction example. Local shared accesses are replaced by privatized memory accesses when it is possible. Such optimization requires the computational block to be split into three parts: computation of the most upper face, computation of the middle chunk of faces, and computation of the lower face. The middle chunk of faces, lines 41 through 43, does not require any remote shared accesses; therefore, it can be fully privatized. For the computations over the boundary blocks, however, this is not the case. The upper and lower faces, lines 28 through 34 and 36 through 40, respectively, need to perform both local and remote shared accesses. Thus, accesses can be only partially optimized. Lines 17 through 27 correspond to the case when each thread is allocated only one face to process. In this case, local and remote shared accesses are also needed. Thus, it is possible to privatize only some of the shared memory accesses.

### *Example 7.1:* heat\_conduction7.upc

```

...
1. // statically allocate grids [] to be matched to [2] [N] [N] [N]
2. shared[BLOCKSIZE] double grids [2] [N] [N] [N];
3. shared double dTmax_local [THREADS];
4. double (*mygrid [2])[N] [N], *mydTmax_ptr;

5. void initialize(void)
6. {

...
7. /* set up private pointers */
8. mygrid [0] = (double (*) [N] [N])
    &grids [0] [N*MYTHREAD/THREADS] [0] [0];
9. mygrid [1] = (double (*) [N] [N])
    &grids [1] [N*MYTHREAD/THREADS] [0] [0];
10. mydTmax_ptr = (double *) &dTmax_local[MYTHREAD];

11. /* sets one edge of the cube to 1.0 (heat) */
12. if (MYTHREAD == 0)
13.     for (y=1; y<N-1; y++)
14.         for (x=1; x<N-1; x++)
15.             mygrid [0] [0] [y] [x] = mygrid [1] [0] [y] [x] = 1.0;
16. }

17. void process_only_face(int sg, int dg, double *dTmax)

```

**130** PERFORMANCE TUNING AND OPTIMIZATION

```
18. { // grid(,z-1,,) and grid(,z+1,,) are remote shared
19.   double T, dT;
20.   int y, x, global_z;

21.   global_z = N*MYTHREAD/THREADS;

   ...
22.   T = (grids[sg][global_z+1][y][x] + grids[sg][global_z-
23.     1][y][x] +
24.     mygrid[sg][0][y+1][x] + mygrid[sg][0][y-1][x] +
25.     mygrid[sg][0][y][x+1] + mygrid[sg][0][y][x-1] ) / 6.0;
26.   dT = T - mygrid[sg][0][y][x];
27.   mygrid[dg][0][y][x] = T;

   ...
27. }

28. void process_front_face(int sg, int dg, double *dTmax)
29. { // grid(,z-1,,) is remote shared, all others are mygrid[ ]

   ...
30.   global_z = N*MYTHREAD/THREADS;
   ...

31.   T =   mygrid[sg][1][y][x] + grids[sg][global_z-1][y][x] +
32.     mygrid[sg][0][y+1][x] + mygrid[sg][0][y-1][x] +
33.     mygrid[sg][0][y][x+1] + mygrid[sg][0][y][x-1] ) / 6.0;

   ...
34. }
35.
36. void process_back_face(int sg, int dg, double *dTmax)
37. { // grid(,z+1,,) is remote shared, all others are mygrid[ ]

   ...
38.   z = N/THREADS-1;
39.   global_z = N*MYTHREAD/THREADS + z;
   ...
40. }

41. void process_interior_face(int z, int sg, int dg,
42.   double *dTmax)
43. { // all accesses are done through mygrid[ ] [ ] [ ] [ ]
   ...
43. }
```

```

44. int heat_conduction(void)
45. {
    ...
46.     do
47.     {
48.         dTmax = 0.0;
49.         if((N/THREADS) == 1)
50.         {
51.             if((MYTHREAD != 0) && (MYTHREAD != (THREADS-1)))
52.                 process_only_face(sg, dg, &dTmax);
53.         }
54.         else
55.         {
56.             if(MYTHREAD != 0)
57.                 process_front_face(sg, dg, &dTmax);
58.             for(z=1; z<(N/THREADS)-1; z++)
59.                 process_interior_face(z, sg, dg, &dTmax);
60.             if(MYTHREAD != (THREADS-1))
61.                 process_back_face(sg, dg, &dTmax);
62.         }
63.         *mydTmax_ptr = dTmax;
64.         upc_barrier;
    ...
65.     } while(finished == 0);
66.     return nr_iter;
67. }

```

### Aggregation of Remote Accesses

As explained earlier, aggregating remote accesses into one large access has the potential of achieving high network bandwidth. The aggregation of remote shared accesses is illustrated in Example 7.2 with our heat conduction problem. Only changes from the previous version of the example are shown. The main modifications have been made to the functions that operate on the boundaries. Lines 25 through 33 have been modified to use the ghost zones information, cached locally at each iteration by the function `update_ghostzones()`, line 44. Thus, all computations are done over data with affinity to the processing thread, thereby avoiding remote accesses. To amortize the communication overhead and improve the bandwidth, the ghost zones are refreshed, lines 5 through 17, using string (or bulk) functions, `upc_memput()` in this case.

*Example 7.2:* heat\_conduction8.upc

```

...
1. #define FRONT 0
2. #define BACK 1
3. shared [2*N*N] double ghostzones[THREADS][2][N][N];
4. double (*front_ghostzone)[N], (*back_ghostzone)[N];

5. void update_ghostzones(int sg)
6. {
7.     if(MYTHREAD != 0)
8.         { // send mygrid[sg][0] to back[MYTHREAD-1]
9.             upc_memput(ghostzones[MYTHREAD-1][BACK],
10.                        &mygrid[sg][0][0][0], sizeof(double)*N*N);
11.         }

12.     if(MYTHREAD != THREADS-1)
13.         { // send mygrid[sg][N/THREADS-1] to front[MYTHREAD+1]
14.             upc_memput(ghostzones[MYTHREAD+1][FRONT],
15.                        &mygrid[sg][N/THREADS-1][0][0], sizeof(double)*N*N);
16.         }
17. }

18. void initialize(void)
19. {
...
20.     front_ghostzone = (double (*)[N])
21.         &ghostzones[MYTHREAD][FRONT][0][0];
22.     back_ghostzone = (double (*)[N])
23.         &ghostzones[MYTHREAD][BACK][0][0];

...
24. }

25. void process_only_face(int sg, int dg, double *dTmax)
26. { // uses back_ghostzone[][] and front_ghostzone[][]
    instead of doing accesses to grids[][][][]

...
27. }

28. void process_front_face(int sg, int dg, double *dTmax)
29. { // uses front_ghostzone[][]

...

```

```

30. }

31. void process_back_face(int sg, int dg, double *dTmax)
32. { // uses back_ghostzone[][]

    ...
33. }

34. int heat_conduction(void)
35. {

    ...
36.     if ((N/THREADS)==1)
37.         { // for this particular case, ensure that initial
              conditions are into ghostzones
38.             update_ghostzones(sg);
39.             upc_barrier;
40.         }

41.     do
42.     {

        ...
43.         upc_barrier;
        ...
44.         update_ghostzones(dg); // dg is the up-to-date
                                   grid[][][]
45.         upc_barrier;

        ...
46.     } while (finished == 0);
47.     return nr_iter;
48. }

```

### Parallel Binary Tree

In this study we present the use of locks to access and update a parallel tree data structure. In the parallel binary tree we use the following data structure for the node and access definitions:

```

struct tree_node {
    shared struct tree_node *up, *left, *right;
    shared void *data;
    upc_lock_t *lock;
};

```



```

shared struct tree_node insert_left
    (shared struct tree_node *node, shared void *data);

shared struct tree_node insert_right
    (shared struct tree_node *node, shared void *data);

upc_lock_t *find_node_lock
    (shared struct tree_node *node);

shared struct tree_node trav_right
    (shared struct tree_node *node);

shared struct tree_node trav_left
    (shared struct tree_node *node);

shared struct tree_node trav_up
    (shared struct tree_node *node);

```

The primitives of this data structure are to splice a new node into either the left or right child of a given node and traverse up, left or right. The `find_node_lock()` routine finds the next-higher node lock, as the density of locks in the tree is a tunable parameter. Start with inserting a new item in a tree node:

```

upc_lock_t *find_node_lock (shared struct tree_node *node)
{
    if (node->lock)
        return lock;
    return find_node_lock (node->up);
}

shared struct tree_node insert_left
    (shared struct tree_node *node, shared void *data)
{
    shared struct tree_node *new;
    upc_lock_t *lock;

    new = upc_local_alloc (1, sizeof struct tree_node);
    new->data = data;
    if (this_node_needs_a_lock)
        new->lock = upc_lock_alloc();
    else
        new->lock = NULL;

    lock = find_node_lock (node);
    upc_lock (lock);
}

```

```

    new->up = node;
    new->left = node->left;
    new->right = 0;
    new->left->up = new;
    node->left = new;
    upc_unlock(lock);

    return new;
}

shared struct tree_node trav_left
    (shared struct tree_node *node)
{
    upc_lock_t *lock;
    shared struct tree_node *tmp;

    lock = find_node_lock (node);
    upc_lock (lock);
    tmp = node->left;
    upc_unlock (lock);
    return tmp;
}

```

## 7.6 SUMMARY

This chapter focuses on optimizing UPC applications. Performance optimization relies on understanding the general computer architecture and programming model issues and trying to write code that masks out the weaknesses and exploits the strengths of modern parallel computers. Many performance optimizations can be performed automatically by compilers and run-time systems. However, some must be done explicitly by the programmer. In a new language such as UPC, it will take time to have compilers that relieve the programmer from the burden of doing hand performance tuning. However, we expect, based on our experiences, that compilers and run-time systems will continue to improve to the point that direct intervention by the programmer will become minimal. Until this happens, programmers should employ hand tunings that can perform self-processing of local data, privatization of space, aggregation and prefetching of accesses, reduced synchronization when possible, and load balancing.

## EXERCISES

**7.1** Integrate the privatization, then the aggregation optimizations, into the split-phase heat conduction program described in Section 6.2.

- 7.2** Measure the wall time taken by heat conduction processing (this is excluding the time for initialization, displaying results, and producing the file output). The wall time is measured using `gettimeofday()`. Time multiple versions of the heat conduction problem as shown in Chapters 5, 6, and 7. Quantify the impact of the optimizations over the execution time and the scalability.
- 7.3** Implement a parallel insert into a distributed binary search tree.

## UPC Libraries

Libraries provide standard toolsets and interfaces to perform common tasks conveniently by programmers while providing developers the opportunity to optimize and improve the implementations of such tasks for efficient executions. The UPC language has two standard libraries that will soon be integrated into the general language specification. These are the collective library and the I/O library. The *collective library* defines a set of collective communications and simple operations that can reduce the development time of common and mundane tasks. The *I/O library* defines parallel operations on files.

Libraries do not only make programming more convenient but also enable high performance. Stating collective and I/O operations as standard application program interfaces (APIs) allows the compiler developers to optimize at a very low level such types of operations for the target architectures. They also provide the programmer with the tools for hiding the latency associated with extensive communications and I/O operations. Full specifications of the collective and I/O libraries are included in Appendixes B and C, respectively. This chapter draws from the material in those specifications to provide a quick summary of the two libraries. Interested readers should refer to these appendixes for more information.

### 8.1 UPC COLLECTIVE LIBRARY

For many applications, UPC provides programmers with the ability to express locality, which can provide for high-performance execution throughout the program run. In other applications, data may be highly shared and perhaps used in different phases of the program by different threads. Thus, a static distribution may not provide what is needed for locality exploitation, and data redistribution may be necessary. In addition, experience has shown that there are a number of common operations that are used by a large percentage of applications and that require great coordination among the threads. These operations have often been referred to as *collective communications operations* or *collective operations* [WEI02].

The UPC specifications are augmented with a collective library that has been approved by the consortium [SEI03]. To define the semantics of these operations, including consistency implications and behavior, each collective operation can be decomposed into three phases: action, notification, and wait. Each thread participating in a collective operation must pass through each of the phases. These phases can be defined as follows. *Action* is the first phase of a collective operation, in which all threads will be computing the results defined for the operation. According to the nature of the task to be done, synchronizations can take place. On the other hand, certain threads might not be doing anything. *Notification* occurs when a thread finishes its share of the work. At that point it notifies all other threads of its completion. *Wait* is the final phase of a collective operation. It occurs when a thread notifies all others about the completion of its share of the work. In this case, the thread can either wait (or not) for all others to notify. Such a barrier synchronization ensures that all threads have completed their part of the collective operation. However, for improved performance, the programmers can override this barrier if the correctness of the result does not require it.

Thus, UPC offers three types of synchronization semantics for both the input and output shared buffers prior to and after each collective function call. By specifying `UPC_IN_{NO, MY, ALL} SYNC` and/or `UPC_OUT_{NO, MY, ALL} SYNC`, the developer, can avoid unnecessary synchronization overheads.

## Categories of Collective Operations

The current UPC v1.1 specifications already incorporate numerous collective operations in the field of shared memory allocation and synchronization, most of which have been discussed in Chapters 5 and 6. The UPC collective library specification augments the UPC language with additional collective functions that target data relocation and computational operations.

In this chapter, only new operations introduced by the collective library are discussed. These operations are focused on two areas: the relocation of data, and common computational functions that integrate data from all threads. *Relocalization* refers to shared data redistributions such that affinity of data is reassigned. This can be useful when locality properties of data are good for one phase of the execution but not for others. There, the UPC collective library is including advanced relocalization functions, as `permute`, `gather`, or `scatter` to provide such capabilities easily to developers. Figures 8.1 through 8.5 illustrate all the relocalization functions available in the UPC collective library.

- The `upc_all_broadcast` function (Figure 8.1) copies a block of memory with affinity to a single thread to a block of shared memory on each thread.
- The `upc_all_scatter` function (Figure 8.2) copies the `i`th block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the `i`th thread.

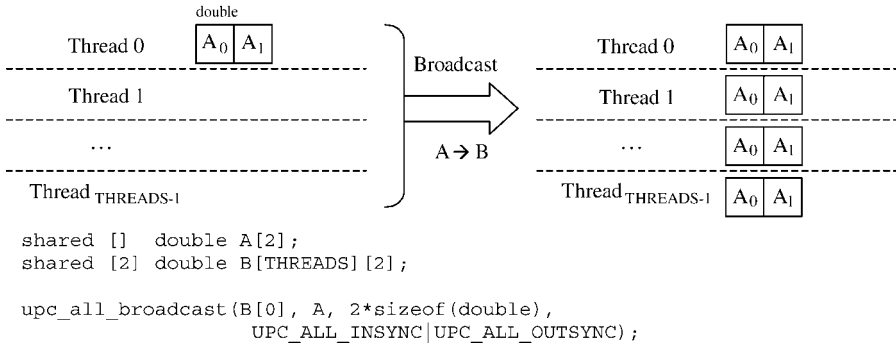


Figure 8.1 Broadcast

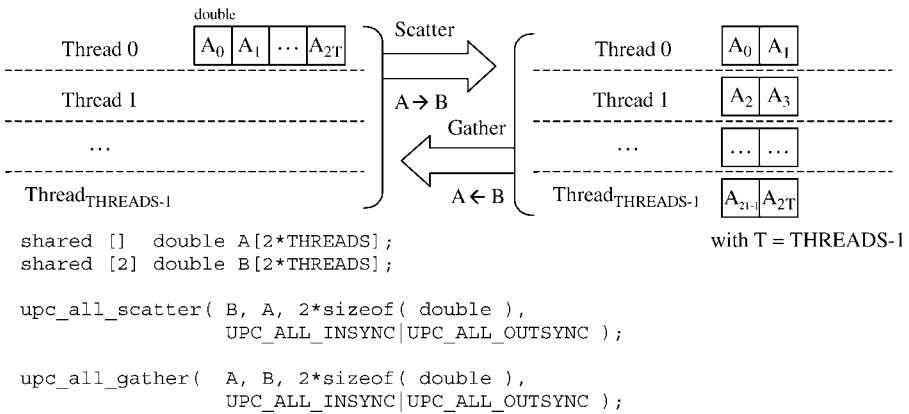


Figure 8.2 Scatter

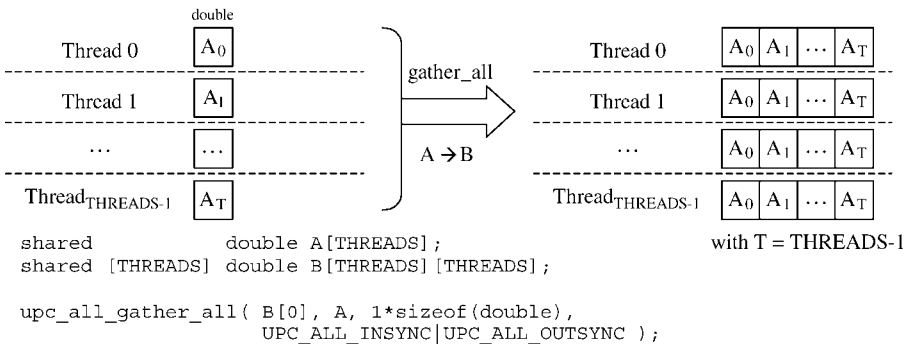


Figure 8.3 Gather to All Threads

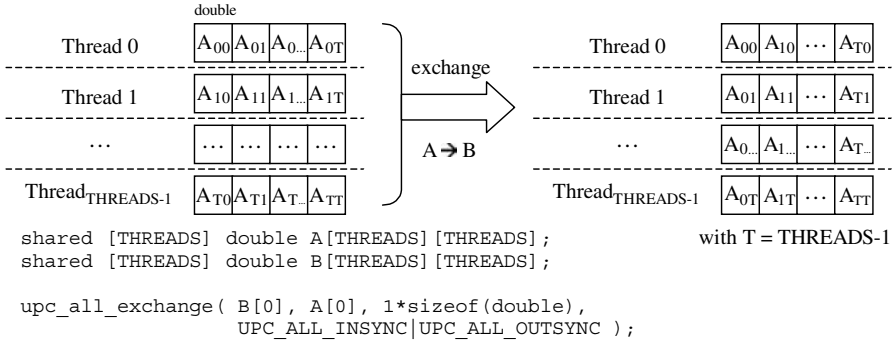


Figure 8.4 Exchange

- The `upc_all_gather` function (Figure 8.3) copies a block of shared memory that has affinity to the *i*th thread to the *i*th block of a shared memory area that has affinity to a single thread.
- The `upc_all_exchange` function (Figure 8.4) copies the *i*th block of memory from a shared memory area that has affinity to thread *j* to the *j*th block of a shared memory area that has affinity to thread *i*.
- The `upc_all_permute` function (Figure 8.5) copies a block of memory from a shared memory area that has affinity to the *i*th thread to a block of a shared memory that has affinity to thread `perm[i]`.

The UPC collective library also provides computational operations, as reduce, prefix reduce, and sort. Figure 8.6 illustrates a reduction operation.

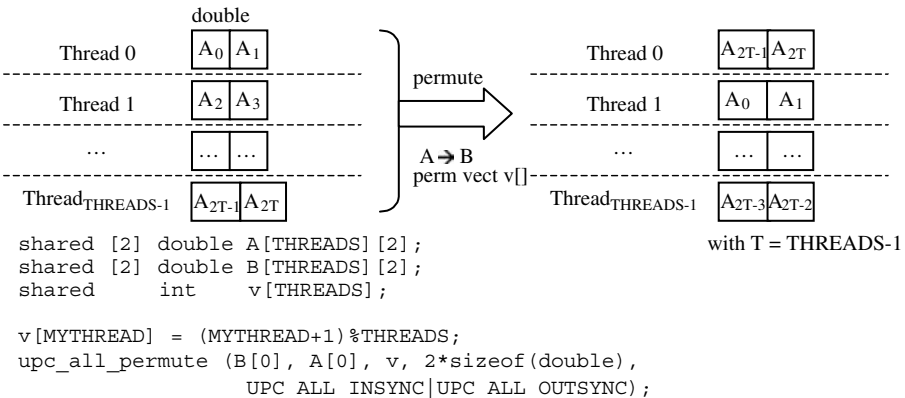
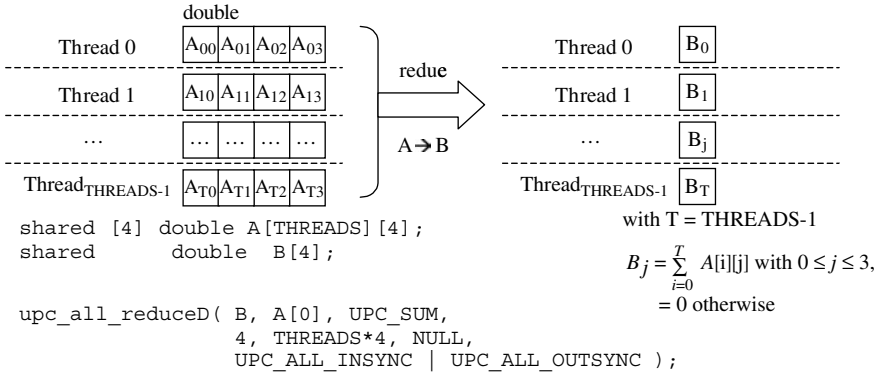


Figure 8.5 Permute



**Figure 8.6** Reduction Collective Operation

## 8.2 UPC-IO LIBRARY

The UPC-IO API [ELG04] 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 library 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 [MPI97].

I/O is known to be a major bottleneck compared with the dramatically increasing processing capabilities. This problem becomes more significant for massively parallel computers where I/O requirements are orders of magnitude higher than conventional computers. Parallel I/O subsystems can provide the ability to increase I/O bandwidth through reading and writing data in parallel. The application software, however, must have access to an I/O API of an optimized I/O library implementation in order to exploit parallel I/O opportunities in the underlying system.

In the absence of an effective parallel I/O API, parallel programs have two options. The first is to allow each executing thread to handle its own file I/O independently. This results in many inefficient and redundant operations. The other option is to delegate the I/O of all threads to only one dedicated thread. This I/O thread can then read and distribute data to other threads that need the data. It can also collect write requests from the other threads and perform them on their behalf. This approach is not efficient and can create a performance bottleneck.

### Parallel I/O in Parallel Programming

From the user's perspective, parallel I/O [INT95, MAY00] can be viewed as the ability to access concurrently (read or write) a single file from multiple threads in an efficient manner. To accomplish this, an API specifically designed for parallel I/O concepts that enable performance optimizations and define the consistency semantics in the presence of concurrent writes is needed. Typical performance optimizations include support for collective I/O (i.e., the semantics to specify how a



group of threads access a file simultaneously) and support for noncontiguous accesses. UPC-IO is designed to embody these features.

### **UPC Perspective of Parallel I/O**

Any parallel I/O interface has to consider several design issues. Some of the design issues and rationale for constructing the I/O API, discussed next, were influenced primarily by the UPC design philosophy and concepts.

### **Library-Based Versus Operating System-Based I/O**

Implementations of parallel I/O in a programming paradigm can be operating system-based or library-based. OS-based interfaces are powerful because of their direct access to low-level operating system services. The problem, however, is that of portability, as operating systems vary from one vendor to another. One example of such an interface is the Intel Paragon File System (PFS) [INT95]. Library-based interfaces, on the other hand, are generally available for multiple platforms. One good example of a library-based I/O interface is MPI-IO [MPI97]. As UPC is designed to run across multiple platforms from different vendors, the clear choice is to design our interface as a library API, which can be implemented and optimized by the respective vendors.

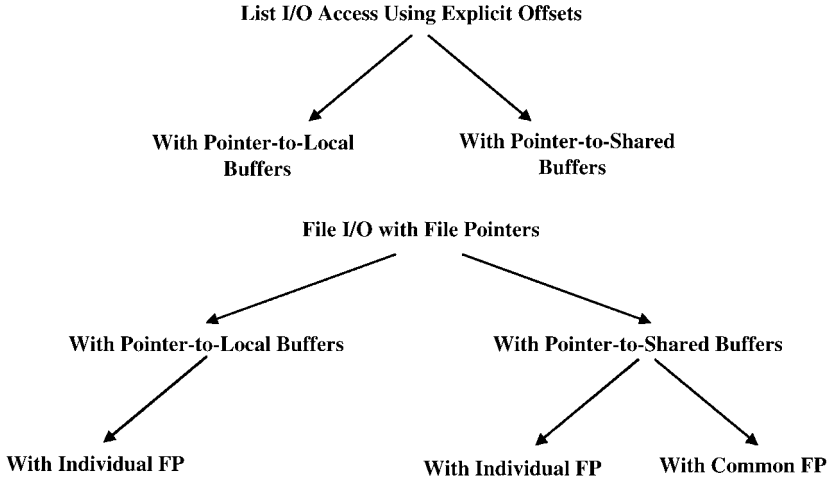
### **Collective I/O Functions**

When multiple threads are involved, file I/O primitives can be called by a single thread or by a group of threads. When multiple threads are calling, the calls could be independent or collective. Furthermore, in a collective call, an I/O library may allow all or only a group of threads to participate. Independent functions allow threads to read or write data at any time and location, without dependencies among the threads. On the other hand, collective file accesses mean that a group of threads, or all threads, read or write data concurrently and in coordination with one another, allowing substantial opportunities for optimizations.

UPC-IO supports only collective I/O functions. That is, all threads must call the UPC-IO functions. In case a thread needs to access a file independently, it can use the regular C/POSIX I/O functions. A file must not, however, be open concurrently by using C/POSIX I/O and UPC-IO. Data consistency is not guaranteed in such a case. Collective I/O operations in UPC-IO are defined over all threads only. This is because the UPC language specification does not support collective operations over a group of threads.

### **File Accessing and File Pointers**

Collective UPC-IO accesses can be done in and out of shared and private buffers; thus, private and shared reads and writes are generally supported. In each of these cases, file pointers could be either common or individual. File pointer modes are



**Figure 8.7** UPC-IO File Access Methods

specified by passing a flag to the collective `upc_all_fopen` function and can be changed using `upc_all_fcntl`. When a common file pointer is used, all threads positioned in the file will be aligned with that common file pointer and based on their thread id. When a file is opened with an individual file pointer flag, each thread gets its own file pointer and advances through the file at its own pace.

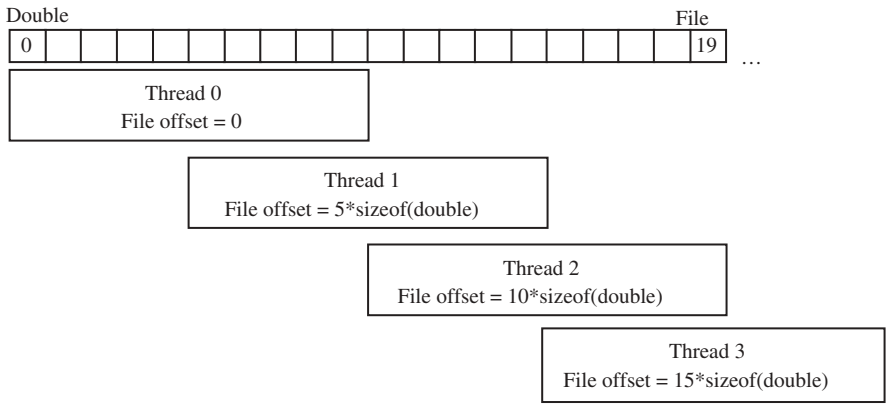
Examples 8.1 through 8.3 and provide examples of UPC-IO functions and their use.

### **Example 8.1**

```

1. double buffer[10]; // and assuming a total of 4 THREADS
2. upc_file_t *fd;
3. fd = upc_all_fopen("file", UPC_RDONLY | UPC_INDIVIDUAL_FP,
                    0, NULL);
4. upc_all_fseek(fd, 5*MYTHREAD*sizeof(double), UPC_SEEK_SET);
5. upc_all_fread_private(fd, buffer, sizeof(double), 10, 0);
6. upc_all_fclose(fd);
  
```

Figure 8.8 represents Example 8.1, which considers a collective read operation. Each thread reads a block of data into a private buffer from a particular thread-specific offset. Figure 8.9 illustrates how the file is viewed in Example 8.1. The data read is stored in a shared buffer having a block size of five elements. The user selects the type of file pointer at file-open time. By default, individual file pointers are used. The user can select the common file pointer by passing the flag `UPC_SHARED_FP` to `upc_all_fopen`.



**Figure 8.8** Collective Read into Private Can Provide Canonical File View

**Example 8.2**

```

1. shared [5] float buffer [20] ; // and assuming a total of 4
                                static THREADS
2. upc_file_t *fd;
3. fd = upc_all_fopen ("file", UPC_RDONLY | UPC_COMMON_FP,
                       0, NULL);
4. upc_all_fread_shared (fd, buffer, 5, sizeof(float), 20,
                        UPC_ALL_SYNC);

```

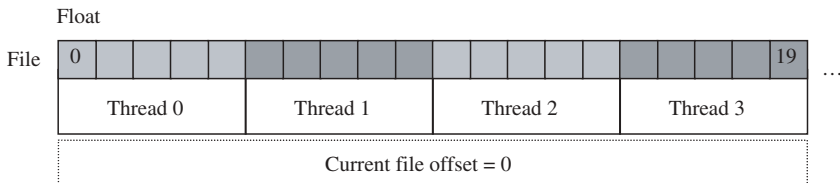
The list I/O functions allow the user to specify noncontiguous accesses in both memory and file in the form of lists of explicit offsets and lengths in the file and explicit address and lengths in memory. None of the file pointers are used or updated in this case. An example of a program that uses list I/O is Example 8.3. The resulting data transfer is illustrated in Figure 8.10.

**Example 8.3**

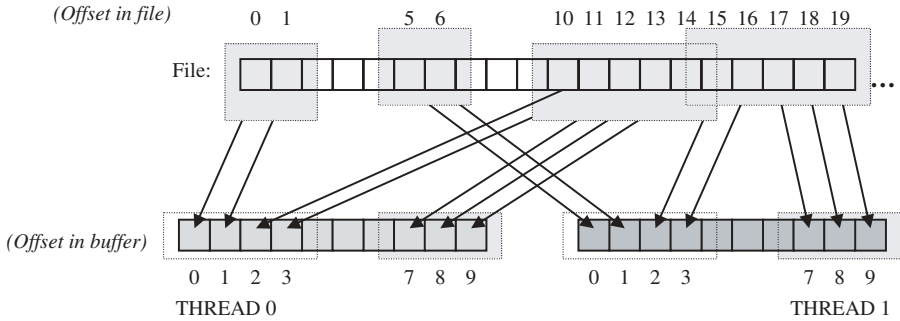
```

1. upc_local_memvec_t memvec [2] ;
2. upc_filevec_t filevec [2] ;
3. upc_file_t *fd;

```



**Figure 8.9** Read into a Blocked Shared Buffer Can Provide a Partitioned File View



**Figure 8.10** List I/O Read of Noncontiguous Parts of a File to Private Noncontiguous Buffers

```

4. char buffer [12] ;
5. fd = upc_all_open ("file",
                      UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL) ;
6. memvec [0] .baseaddr = &buffer [0] ;
7. memvec [0] .len = 4 ;
8. memvec [1] .baseaddr = &buffer [7] ;
9. memvec [1] .len = 3 ;
10. filevec [0] .offset = MYTHREAD* 5 ;
11. filevec [0] .len = 2 ;
12. filevec [1] .offset = 10+MYTHREAD* 5 ;
13. filevec [1] .len = 5 ;
14. upc_all_fread_list_local (fd, 2, &memvec, 2, &filevec,
                              UPC_OUT_ALLSYNC) ;

```

## Synchronous and Asynchronous I/O

I/O operations can be synchronous (blocking) or asynchronous (nonblocking). Whereas synchronous calls are quite simple and easy to use from a programming point of view, asynchronous operations allow the overlapping of computation and I/O to achieve improved performance. Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an asynchronous call starts an I/O operation and returns immediately. Thus, the executing process can turn its attention to other processing needs while the I/O is progressing.

UPC-IO supports both synchronous and asynchronous I/O functions. The asynchronous I/O functions have the same syntax as their synchronous counterparts, with the addition of “*async*” in their names. The asynchronous I/O functions have the restriction that only one (collective) asynchronous operation can be active at a time on a given file descriptor. That is, an asynchronous I/O function must be completed by calling `upc_all_ftest_async` or `upc_all_fwait_async` before another asynchronous I/O function can be called on the same file descriptor.

## Consistency and Atomicity Semantics

The consistency semantics define when the data written to a file by a thread is visible to other threads. The atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file and some of the writes overlap each other. For performance reasons, UPC-IO, by default, supports weak consistency and atomicity semantics. The user can select strong semantics by passing the flags `UPC_STRONG_CA` to `upc_all_fopen`.

The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using private file pointers or list I/O functions. By default, the data in the overlapping space is undefined. If the user selects the `UPC_STRONG_CA` mode, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order. In that case, the atomicity is guaranteed at the UPC-IO function level. The entire operation specified by a single function is performed atomically, regardless of whether it represents a single, contiguous read/write or multiple noncontiguous reads or writes as in a list I/O function.

## 8.3 SUMMARY

The UPC programming language embodies two standard libraries, one for collective operations and the other for parallel I/O. The collective operations library focuses on data relocalization, which is to redistribute the data or reassign its affinity. These operations are quite helpful when the initial data distribution cannot effectively serve all phases of the application, and each phase can benefit from a different distribution. The collective operations library also includes mathematical functions that can help integrate results from the computing threads. The I/O library supports collective reads and writes into and from a file effectively. It provides for contiguous as well as list I/O. Both synchronous and asynchronous versions of the calls are provided in order to hide IO latencies when possible.

## REFERENCES

---

- [ALV90] Alverson, Robert; Callahan, David; Cummings, Daniel; Koblenz, Brian; Porterfield, Allan; Smith, Burton; The Tera computer system, *Proceedings of the 4th International Conference on Supercomputing*, Amsterdam, The Netherlands, June 11–15, 1990, pp. 1–6.
- [BUT97] Butenhof, David R.; *Programming with POSIX Threads*, Addison-Wesley, Reading, MA, 1997.
- [CAR99] Carlson, William; Draper, Jesse; Culler, David; Yelick, Kathy; Brooks, Eugene; Warren, Karen; *Introduction to UPC and Language Specification*, CCS-TR-99-157, IDA/CCS, Bowie, MD, May 1999.
- [CUL93] Culler, David E.; Dusseau, Andrea; Goldstein, Seth Copen; Krishnamurthy, Arvind; Lumetta, Steven; von Eicken, Thorsten; Yelick, Katherine; Parallel programming in Split-C, *Proceedings of Supercomputing '93*, November 1993, pp. 262–273.
- [DIJ71] Dijkstra, Edsger W., *Hierarchical Ordering of Sequential Processes*, Acta Informatica 1, 115–138, Springer-Verlag, 1971.
- [ELG01] El-Ghazawi, Tarek; Carlson, William; Draper, Jesse; *UPC Language Specifications v1.0*, <http://upc.gwu.edu>, February 2001.
- [ELG03] El-Ghazawi, Tarek; Carlson, William; Draper, Jesse; *UPC Language Specifications v1.1*, <http://upc.gwu.edu>, October 2003.
- [ELG04] El-Ghazawi, Tarek; Cantonnet, Francois; Saha, Proshanta; Thakur, Rajeev; Ross, Rob; Bonachea, Dan; *UPC-IO: A Parallel I/O API for UPC v1.0*, July 2004, <http://upc.gmu.edu>.
- [HPF97] High Performance Fortran Forum; *High Performance Fortran Language Specification*, version 2.0, January 1997, <http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm>.
- [HU00] Hu, Y. Charlie; Lu, Honghui; Cox, Alan L.; Zwaenepoel, Willy; OpenMP for networks of SMPs, *Journal of Parallel and Distributed Computing*, Vol. 60, No. 12, pp. 1512–1530, December 2000.
- [HWA98] Hwang, Kai; Xu, Zhiwei; *Scalable Parallel Computing: Technology, Architecture, Programming*, WCB/McGraw-Hill, New York, 1998.
- [INT95] Intel Scalable Systems Division, *Paragon System User's Guide*, Order Number 312489-004, May 1995, Hillsboro, OR.
- [ISO99] ISO/IEC 9899:1999, *Programming Languages—C*, ISO/IEC, December 1999, Geneva, Switzerland.

- [LAM74] Lamport, Leslie; A new solution of Dijkstra's concurrent programming problem, *Communications of the ACM*, Vol. 17, No. 8, pp. 453–455, 1974.
- [MAY00] May, John; *Parallel I/O for High Performance Computing*, Morgan Kaufmann, San Francisco, October 2000.
- [MPI94] Message Passing Interface Forum; MPI: a message-passing interface standard, *International Journal of Supercomputer Applications*, Vol. 8, No. 3/4, pp. 157–416, 1994, special issue on MPI.
- [MPI97] Message Passing Interface Forum; *MPI-2: Extensions to the Message-Passing Interface*, July 18, 1997, <http://www.mpi-forum.org>.
- [NAS97] Nastea, Sorin G.; Frieder, Ophir; El-Ghazawi, Tarek; Load-balanced sparse matrix–vector multiplication on parallel computers, *Journal of Parallel and Distributed Computing*, Vol. 46, No. 2, pp. 180–193, 1997.
- [NPB03] NASA Advanced Supercomputing; *NAS Parallel Benchmark Suite*, <http://www.nasa.gov/Software/NPB>, 2003.
- [NTM04] Combined research and curriculum development; nontraditional manufacturing (NTM), *Modeling of Laser Machining Process*, Chapter 3.5, Columbia University, <http://www.columbia.edu/cu/mechanical/mrl/ntm/pgIndex.html>, June 27, 2004.
- [OPE97] OpenMP Standards Boards; *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, <http://www.openmp.com>, October 1997.
- [QUA03] Quadrics Ltd.; *Elan Programming Manual*, Edition 5.1, Quadrics, November 2003, Bristol, UK.
- [ROS87] Rose, John R.; Steele, Guy L.; C\*: an extended C language for data parallel programming, Lana P. and Steven I. Kartashev, editors, *Proceedings of the 2nd International Conference on Supercomputing*, Vol. H, International Supercomputing Institute, Santa Clara, CA, May 1987, pp. 2–16.
- [SEI03] Seidel, Steven; Greenberg, David; Wiebel, Elisabeth; *UPC Collective Operations Specifications v1.0*, [http://www.gwu.edu/upc/docs/UPC\\_Coll\\_Spec\\_V1.0.pdf](http://www.gwu.edu/upc/docs/UPC_Coll_Spec_V1.0.pdf) December 2003.
- [SNI98] Snir, Marc; Gropp, William; *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1998.
- [STE01] Sterling, Thomas; *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA, 2001.
- [WEI02] Wiebel, Elisabeth; Greenberg, David; *Collective Operations in UPC*, draft, August 2002.

# UPC Language Specifications, v1.1.1

Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper

## Introduction

- 1 Scope
- 2 Normative References
- 3 Terms, Definitions, and Symbols
  - 3.1 Access
    - 3.1.1 Local Access
    - 3.1.2 Private Access
    - 3.1.3 Remote Access
  - 3.2 Affinity
  - 3.3 Shared Object
  - 3.4 Pointer-to-Shared
  - 3.5 Thread
  - 3.6 Collective
  - 3.7 Single-Valued
  - 3.8 Phase
- 4 Conformance
- 5 Environment
  - 5.1 Conceptual Models
    - 5.1.1 Translation Environment
    - 5.1.2 Execution Environment
- 6 Language
  - 6.1 Notation
  - 6.2 Predefined Identifiers
    - 6.2.1 THREADS

---

*UPC: Distributed Shared Memory Programming*, by Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick  
Published 2005 John Wiley & Sons, Inc.



- 6.2.2 MYTHREAD
- 6.2.3 UPC\_MAX\_BLOCK\_SIZE
- 6.3 Expressions
  - 6.3.1 upc\_localsizeof Operator
  - 6.3.2 upc\_blocksizeof Operator
  - 6.3.3 upc\_elemsizeof Operator
  - 6.3.4 Pointer-to-Shared Arithmetic
  - 6.3.5 Cast and Assignment Expressions
  - 6.3.6 Address Operators
- 6.4 Declarations
  - 6.4.1 Type Qualifiers
  - 6.4.2 Shared and Reference Type Qualifiers
  - 6.4.3 Declarators
- 6.5 Statements and Blocks
  - 6.5.1 Barrier Statements
  - 6.5.2 Iteration Statements
- 6.6 Preprocessing Directives
  - 6.6.1 UPC Pragmas
  - 6.6.2 Predefined Macro Names
- 7 Library
  - 7.1 Standard Headers
  - 7.2 UPC Utilities <upc.h>
    - 7.2.1 Termination of All Threads
    - 7.2.2 Shared Memory Allocation Functions
    - 7.2.3 Pointer-to-Shared Manipulation Functions
    - 7.2.4 Lock Functions
    - 7.2.5 Shared String-Handling Functions

Acknowledgments

References

## INTRODUCTION

1. UPC is a parallel extension to the C Standard. UPC follows the distributed shared memory programming paradigm. The first version of UPC, known as version 0.9, was published in May 1999 as a technical report [1] at the Institute for Defense Analyses Center for Computing Sciences.

2. Version 1.0 of UPC was discussed initially at the UPC workshop held in Bowie, Maryland, May 18–19, 2000. The workshop had about 50 participants from industry, government, and academia. This version was adopted with modifications in the UPC miniworkshop meeting held during Supercomputing 2000 in November 2000 in Dallas, and finalized in February 2001.
3. Version 1.1 of UPC was discussed initially at the UPC workshop held in Washington, DC, March 3–5, 2002, and finalized in March 2003.

**1 SCOPE**

1. This document focuses on the UPC specifications that extend the C Standard to an explicit parallel C based on the distributed shared memory model. All C specifications from ISO/IEC 9899 [2] are considered a part of these UPC specifications, so are not addressed in this document.
2. Small parts of the C Standard [2] may be repeated for self-containment and clarity of a subsequent UPC extension definition.

**2 NORMATIVE REFERENCES**

1. The following document and its identified normative references in addition to these documents constitute provisions of these UPC specifications. This will not apply to subsequent revisions of this document.

**Table 1 Mapping UPC Sections to C Standard Specifications Sections**

UPC Specifications Section	C Standard Specifications Section
1	1
2	2
3	3
4	4
5	5
6	6
6.1	6.1
6.2	6.4.2.2
6.3	6.5
6.3.6	6.5.3.1
6.4	6.7
6.4.1	6.7.3
6.4.3	6.7.5
6.5	6.8
6.6	6.10
7	7
7.1	7.1.2

2. ISO/IEC 9899: 1999(E), *Programming Languages—C* [2].
3. The section numbering of this document is consistent with the previous document [2]. The correspondence between the section of this document and the previous document is given in Table 1.
4. In the beginning of each UPC specifications section, the corresponding C Standard [2] section is noted.

### 3 TERMS, DEFINITIONS, AND SYMBOLS

1. For the purpose of these specifications, the following definitions apply.
2. Other terms are defined where they appear in *italic* type or on the left-hand side of a syntactical rule.

#### 3.1 Access

Access is the <execution-time action> to read or modify the value of an object by a thread.

**3.1.1 Local Access** This is the <execution-time action> to read or modify, by a given thread, the value of an object in either the private space of that thread or in the shared address locations that have affinity to that thread.

**3.1.2 Private Access** This is the <execution-time action> to read or modify, by a given thread, the value of an object in the private address space of that thread.

**3.1.3 Remote Access** This is the <execution-time action> to read or modify, by a given thread, the value of an object whose address is in the shared address space portion that has affinity to the other threads.

#### 3.2 Affinity

Affinity is a logical association of a portion of the shared address space with a given thread.

#### 3.3 Shared Object

A shared object resides in the shared address space.

#### 3.4 Pointer-to-Shared

Pointer-to-shared denotes a pointer to a shared object.

### 3.5 Thread

A thread is a program task in execution with access not only to a private memory space, but also to a shared memory space that can be accessed by other threads.

### 3.6 Collective

A collective is a requirement placed on some language operations that constrains evaluation of such operations to be matched<sup>1</sup> across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.

### 3.7 Single-Valued

A single-valued operand to a collective operation that has the same value on every thread. The behavior of the operation is otherwise undefined.

### 3.8 Phase

In a phase, an unsigned integer value is associated with a pointer-to-shared which indicates the element offset within an affinity block; used in pointer-to-shared arithmetic to determine affinity boundaries.

## 4 CONFORMANCE

1. In this document, “shall” is to be interpreted as a requirement on a UPC implementation; conversely, “shall not” is to be interpreted as a prohibition.
2. If a “shall” or “shall not” requirement of a constraint is violated, the behavior will be undefined. Undefined behavior is indicated by “undefined behavior” or by the omission of any explicit definition of behavior from the UPC specification.

## 5 ENVIRONMENT

### 5.1 Conceptual Models

#### 5.1.1 Translation Environment

**5.1.1.1 Threads Environment** A UPC program is translated under either a “static THREADS” environment or a “dynamic THREADS” environment. Under

<sup>1</sup>A collective operation need not provide any actual synchronization between threads unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely on such unspecified synchronization for correctness.

the static `THREADS` environment, the number of threads to be used in execution is indicated to the translator in an implementation-defined manner. If the actual execution environment differs from this number of threads, the behavior of the program is undefined.

### 5.1.2 Execution Environment

1. This section provides the UPC parallel extensions of Section 5.1.2 of [2].
2. Each thread has local data storage on which it operates and which is logically divided into a private portion and a shared portion. All accesses to objects in the private portion of data storage are exactly as described in [2].
3. Each thread may access shared data that has affinity to any thread; the semantics of these accesses are described herein.
4. There is an implicit `upc_barrier` at program startup and termination. Except as specified explicitly by `upc_barrier` operations or by certain library functions (all of which are documented explicitly), there are no other barrier synchronization guarantees among the threads.

*Forward reference:* `upc_barrier` (6.5.1).

**5.1.2.1 Program Startup** In the execution environment of a UPC program, derived from the hosted environment as defined in the C Standard [2], each thread calls the UPC program's `main()` function.<sup>2</sup>

#### 5.1.2.2 Program Termination

1. A program is terminated by the termination of all the threads<sup>3</sup> or a call to the function `upc_global_exit()`.
2. Thread termination follows the C Standard definition of program termination in Section 5.1.2.2.3 of [2]. A thread is terminated by reaching the `}` that terminates the main function, by a call to the `exit` function, or by a return from the initial main. Note that thread termination does not imply the completion of all I/O and that shared data allocated by a thread either statically or dynamically shall not be freed before UPC program termination.

*Forward reference:* `upc_global_exit` (7.2.1).

#### 5.1.2.3 Program Execution

1. References to shared objects shall be either strict or relaxed. For relaxed references there is no change to the C Standard execution model as applied to

<sup>2</sup>For example, in the program `main(){printf("hello");}`, each thread prints `hello`.

<sup>3</sup>A barrier is inserted automatically at thread termination.

an individual thread. This implies that translators are free to reorder and/or ignore operations (including shared operations) as long as the restrictions found in Section 5.1.2.3 of [2] are observed.

2. A further restriction applies to strict references. For each strict reference, the restrictions found in Section 5.1.2.3 of [2] must be observed with respect to all threads if that reference is eliminated (or reordered with respect to all other shared references in its thread).
3. Equally, the behavior of strict shared references can be defined as follows. Label each access to a shared object  $S(i, j)$  or  $R(i, j)$ , where  $S$  represents a strict shared access (read or modify),  $R$  represents a relaxed shared access (read or modify),  $i$  is the thread number making the access, and  $j$  is an integer that increases monotonically as evaluation of the program (in the abstract machine) proceeds from startup through termination. The “abstract order” is a partial ordering of all accesses by all threads such that an access  $x(a, b)$  occurs before  $y(c, d)$  in the ordering if  $a==c$  and  $b < d$ . The “actual order ( $k$ )” for thread  $k$  is another partial order in which  $x(a, b)$  occurs before  $y(c, d)$  if thread  $k$  observes the  $x$  access before it observes the  $y$  access. A thread observes all accesses present in the abstract order which affect either the data written to files by it or its input and output dynamics as described in Section 5.1.2.3 of [2]. The least requirements on a conforming implementation are that:

- $x(a, b)$  must “occur before”  $y(c, d)$  in actual order ( $e$ ) if  $a==c$  and  $a==e$  and  $b < d$
- $x(a, b)$  must “occur before”  $y(c, d)$  in actual order ( $e$ ) if  $a==c$  and  $b < d$  and  $((x==S) \text{ or } (y==S))$ , for all  $e$

unless such a restriction has no effect on either the data written into files at program termination OR the input and output dynamics requirements described in Section 5.1.2.3 of [2].

## 6 LANGUAGE

### 6.1 Notation

1. In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic* type, and literal words and character set members (terminals) by **bold** type. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words “one of.” An optional symbol is indicated by the subscript “opt,” so that

$$\{expression_{opt}\}$$

indicates an optional expression enclosed in braces.

2. When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

## 6.2 Predefined Identifiers

This section provides the UPC parallel extensions of Section 6.4.2.2 of [2].

**6.2.1** `THREADS` `THREADS` is a value of type `int`; it specifies the number of independent computational units and has the same value on every thread. Under the static `THREADS` translation environment, `THREADS` is an integer constant suitable for use in `#if` preprocessing directives.

**6.2.2** `MYTHREAD` `MYTHREAD` is a value of type `int`; it specifies the unique thread index. The range of possible values is  $0 \dots \text{THREADS}-1$ .<sup>4</sup>

**6.2.3** `UPC_MAX_BLOCK_SIZE` `UPC_MAX_BLOCK_SIZE` is a predefined integer constant value. It indicates the maximum value<sup>5</sup> allowed in a layout qualifier for shared data. It shall be suitable for use in `#if` preprocessing directives.

## 6.3 Expressions

This section provides the UPC parallel extensions of Section 6.5 of [2].

### 6.3.1 `upc_localsizeof` Operator

`upc_localsizeof unary-expression`  
`upc_localsizeof (type-name)`

**Constraints** The `upc_localsizeof` operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the `sizeof` operator [2, Section 6.5.3.4] also apply to this operator.

#### *Semantics*

1. The `upc_localsizeof` operator returns the size, in bytes, of the local portion of its operand, which may be a shared object or a shared-qualified type. It returns the same value on all threads; the value is the maximum of the size allocated to objects with affinity to any single thread. The result of `upc_localsizeof` is a compile-time constant.

<sup>4</sup>For example, the program `main() { printf("%d", MYTHREAD); }` prints the numbers 0 through `THREADS-1`, in some order.

<sup>5</sup>For example, `shared [ UPC_MAX_BLOCK_SIZE + 1 ] char x[ UPC_MAX_BLOCK_SIZE + 1 ]` and `shared [*] char x[ (UPC_MAX_BLOCK_SIZE + 1)* THREADS ]` are compile errors.

2. The type of the result is `size_t`.
3. If the operand is an expression, that expression is not evaluated.

### 6.3.2 `upc_blocksizeof` Operator

```
upc_blocksizeof unary-expression
upc_blocksizeof (type-name)
```

**Constraints** The `upc_blocksizeof` operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the `sizeof` operator [2, Section 6.5.3.4] also apply to this operator.

#### *Semantics*

1. The `upc_blocksizeof` operator returns the block size of the operand, which may be a shared object or a shared-qualified type. The block size is the value specified in the layout qualifier of the type declaration. If there is no layout qualifier, the block size is 1. The result of `upc_blocksizeof` is a compile-time constant.
2. If the operand of `upc_blocksizeof` has indefinite block size, the value of `upc_blocksizeof` is 0.
3. The type of the result is `size_t`.
4. If the operand is an expression, that expression is not evaluated.

*Forward reference:* indefinite block size (6.4.2).

### 6.3.3 `upc_elemsizeof` Operator

```
upc_elemsizeof unary-expression
upc_elemsizeof (type-name)
```

**Constraints** The `upc_elemsizeof` operator shall apply only to shared-qualified expressions or shared-qualified types. All constraints on the `sizeof` operator [2, Section 6.5.3.4] also apply to this operator.

#### *Semantics*

1. The `upc_elemsizeof` operator returns the size, in bytes, of the highest-level (leftmost) type that is not an array. For nonarray objects, `upc_elemsizeof` returns the same value as `sizeof`. The result of `upc_elemsizeof` is a compile-time constant.
2. The type of the result is `size_t`.
3. If the operand is an expression, that expression is not evaluated.



### 6.3.4 *Pointer-to-Shared Arithmetic*

1. When an expression that has integer type is added to or subtracted from a pointer-to-shared, the result has the type of the pointer-to-shared operand. If the pointer-to-shared operand points to an element of a shared array object and the shared array is large enough, the result points to an element of the shared array. If the shared array is declared with indefinite block size, the result of the pointer-to-shared arithmetic is identical to that described for normal C pointers in Section 6.5.6 of [2], except that the thread of the new pointer shall be the same as that of the original pointer and the phase component is defined always to be zero. If the shared array has a definite block size, the following example describes pointer arithmetic:

```
shared [B] int *p, *p1; /* B a positive integer */ int i;

p1 = p + i;
```

2. After this assignment, the following equations must hold in any UPC implementation. In each case the `div` operator indicates integer division rounding toward negative infinity, and the `mod` operator returns the nonnegative remainder<sup>6</sup>:

```
upc_phaseof(p1) == (upc_phaseof(p) + i) mod B
upc_threadof(p1) == (upc_threadof(p)
                    + (upc_phaseof(p) + i) div B) mod THREADS
```

3. In addition, the correspondence between shared and private addresses and arithmetic is defined using the following constructs:

```
T *P1, *P2;
shared T *S1, *S2;

P1 = (T*) S1; /* legal if S1 has affinity to MYTHREAD */
P2 = (T*) S2; /* legal if S2 has affinity to MYTHREAD */
```

4. For all `S1` and `S2` that point to two distinct elements of the same shared array object which have affinity to the same thread:
  - `S1` and `P1` shall point to the same object.
  - `S2` and `P2` shall point to the same object.
  - The expression  $((\text{upc\_addrfield}(S2) - \text{upc\_addrfield}(S1))$  shall evaluate to the same value as  $((P2 - P1) * \text{sizeof}(T))$ .

<sup>6</sup>The C “%” and “/” operators do not have the necessary properties.

- If  $S1 < S2$ , then `upc_addrfield(S1)` shall be  $<$  `upc_addrfield(S2)` otherwise `upc_addrfield(S1)` shall be  $>$  `upc_addrfield(S2)`.
5. Two compatible pointers-to-shared that point to the same object (i.e., having the same address and thread components) shall compare as equal according to `==` and `!=`, regardless of whether the phase components match.

*Forward References:* `upc_threadof` (7.2.3.1), `upc_phaseof` (7.2.3.2), `upc_addrfield` (7.2.3.4).

### 6.3.5 Cast and Assignment Expressions

#### Constraints

1. A shared type qualifier shall not appear in a type cast of an object that is not shared-qualified, with the exception of the null pointer-to-shared.<sup>7</sup>
2. The cast of a pointer-to-shared to a pointer-to-private by a thread not having affinity with the referenced object has an undefined result.

#### Semantics

1. The casting or assignment from one pointer-to-shared to another in which either the type size or block size differs results in a pointer with a zero phase unless one of the types is “shared void \*,” the generic pointer-to-shared.
2. If a generic pointer-to-shared is cast to a nongeneric pointer-to-shared type with indefinite block size or with block size of 1, the result is a pointer with a phase of zero. Otherwise, if the phase of the former pointer value is not within the range of possible phases of the latter pointer type, the result is undefined.
3. If a pointer-to-shared is cast<sup>8</sup> to a pointer-to-private<sup>9</sup> and the affinity of the shared data is not to the current thread, the result is undefined.
4. After the assignment

```
shared[B] T *s;
```

```
s = 0;
```

`s` is a null pointer-to-shared,<sup>10</sup> and the operators `upc_threadof(s)` and `upc_phaseof(s)` evaluate to zero for all block sizes `B`.

<sup>7</sup>That is, pointers-to-private cannot be cast to pointers-to-shared.

<sup>8</sup>As such pointers are not type compatible, explicit casts are required.

<sup>9</sup>References through such cast pointers behave exactly as if they were accesses to private objects.

<sup>10</sup>Sections 6.3.2.3 and 6.5.16.1 of [2] imply that an implicit cast is allowed for zero and that all null pointers-to-shared compare equal.

### 6.3.6 Address Operators

*Semantics* When the unary & is applied to a shared structure element of type T, the result has type `shared[ ] T*`.

## 6.4 Declarations

UPC extends the declaration ability of C to allow shared types, shared data layout across threads, and ordering constraint specifications.

### Constraints

1. The declaration specifiers in a given declaration shall not include, either directly or through one or more typedefs, both strict and relaxed.
2. The declaration specifiers in a given declaration shall not specify more than one block size, either directly or indirectly through one or more typedefs.

### Syntax

1. The following is the declaration definition according to Section 6.7 of [2], repeated here for self-containment and clarity of the subsequent UPC extension specifications.
2. `declaration:`  
`declaration-specifiers init-declarator-listopt;`
3. `declaration-specifiers:`  
`storage-class-specifier declaration-specifiersopt`  
`type-specifier declaration-specifiersopt`  
`type-qualifier declaration-specifiersopt`  
`function-specifier declaration-specifiersopt`
4. `init-declarator-list:`  
`init-declarator`  
`init-declarator-list, init-declarator`
5. `init-declarator:`  
`declarator`  
`declarator = initializer`

*Forward reference:* strict and relaxed type qualifiers (6.4.2).

**6.4.1 Type Qualifiers** This section provides the UPC parallel extensions of Section 6.7.3 of [2].

## Syntax

```

type-qualifier:
    const
    restrict
    volatile
    shared-type-qualifier
    reference-type-qualifier

```

### 6.4.2 Shared and Reference Type Qualifiers

#### Syntax

1. shared-type-qualifier:
 

```

shared layout-qualifieropt

```
2. reference-type-qualifier:
 

```

relaxed
strict

```
3. layout-qualifier:
 

```

[constant-expressionopt]
[*]

```

#### Constraints

1. A reference type qualifier shall appear in a qualifier list only when the list also contains a shared type qualifier.
2. A shared type qualifier can appear anywhere that a type qualifier can appear except that it shall not appear in the specifier qualifier list of a structure declaration unless it qualifies a pointer type.
3. A layout qualifier of [\*] shall not appear in the declaration specifiers of a pointer.
4. A layout qualifier shall not appear in the type qualifiers for a pointer to void type.

#### Semantics

1. An object that has shared-qualified type is a shared object.
2. References to shared objects, either directly or via pointer-to-shared indirection, shall be either strict or relaxed. Strict and relaxed references behave as described in Section 5.1.2.3 of this document.
3. A reference shall be determined to be strict or relaxed as follows. If the referenced type is strict-qualified or relaxed-qualified, the reference shall be strict or relaxed, respectively. Otherwise, the reference shall be determined to

be strict or relaxed by the UPC pragma rules, as described in Section 6.6.1 of this document.

4. The layout qualifier dictates the blocking factor for the type being shared qualified. This factor is the nonnegative number of consecutive elements (when evaluating pointer-to-shared arithmetic and array declarations) that have affinity to the same thread. If the optional constant expression is 0 or is not specified, all objects have affinity to the same thread. If there is no layout qualifier, the blocking factor has the default value (1). The blocking factor is also referred to as the block size.
5. A layout qualifier indicating that all array elements have affinity to the same thread is said to specify indefinite block size.
6. The block size is a part of the type compatibility.<sup>11</sup>
7. A shared void\* pointer is assignment compatible with any pointer-to-shared type.
8. If the layout qualifier is of the form “[\*],” the shared object is distributed as if it had a block size of

$$(\text{sizeof}(a) / \text{upc\_elemsizeof}(a) + \text{THREADS} - 1) / \text{THREADS},$$

where “a” is the array being distributed.

9. *Example 1:* declaration of a shared scalar:

```
strict shared int y;
```

strict shared is the type qualifier.

10. *Example 2:* automatic storage duration:

```
void foo(void) {
  shared int x; /* a shared automatic variable is not allowed */
  shared int* y; /* a pointer to shared is allowed */
  int * shared z; /* a shared automatic variable is not allowed */
  ...}
```

11. *Example 3:* inside a structure:

```
struct foo {
  shared int x; /* this is not allowed */
  shared int* y; /* a pointer to a shared object is allowed */
};
```

<sup>11</sup> This is a powerful statement which allows, for example, that in an implementation `sizeof(shared int*)` may differ from `sizeof(shared [10] int*)`, and if T and S are pointer-to-shared types with different block sizes, then T\* and S\* cannot be aliases.

*Forward references:* shared array (6.4.3.2), pointer declarator (6.4.3.1).

### 6.4.3 Declarators

#### Syntax

1. The following is the declarator definition according to Section 6.7.5 of [2], repeated here for self-containment and clarity of the subsequent UPC extension specifications.
2. declarator:
 

```
pointeropt direct-declarator
```
3. direct-declarator:
 

```
identifier
(declarator )
direct-declarator [type-qualifier-listopt assignment-expressionopt]
direct-declarator [static type-qualifier-listopt assignment-expression]
direct-declarator [type-qualifier-list static assignment-expression]
direct-declarator [type-qualifier-listopt *]
direct-declarator (parameter-type-list)
direct-declarator (identifier-listopt)
```
4. pointer:
 

```
*type-qualifier-listopt
*time-qualifier-listopt pointer
```
5. type-qualifier-list:
 

```
type-qualifier
type-qualifier-list type-qualifier
```

#### Constraints

1. No type qualifier list shall specify more than one block size, either directly or indirectly, through one or more typedefs.<sup>12</sup>
2. No type qualifier list shall include both `strict` and `relaxed`, either directly or indirectly, through one or more typedefs.
3. `shared` shall not appear in a declarator that has automatic storage duration unless it qualifies a pointer type.

<sup>12</sup>Although layout qualifiers are seen most often in array or pointer declarators, they are legal in all declarators. For example, `shared[ 3] int y` is a legal declarator.

*Semantics*

1. All static nonarray shared-qualified objects have affinity with thread zero.
2. Only pointer type members of a structure or union may be shared-qualified.<sup>13</sup>

**6.4.3.1 Pointer Declarators** This section provides the UPC parallel extensions of Section 6.7.5.1 of [2].

*Semantics*

1. A shared reference that is cast to nonshared will lose all qualities pertaining to being shared.
2. Shared objects with affinity to a given thread can be accessed by either pointers-to-shared or pointers-to-private of that thread.
3. *Example:*

```
int i, *p;
shared int *q;
q = (shared int *)p; /* is not allowed */
if (upc_threadof(q) == MYTHREAD)
    p = (int *) q; /* is allowed */
```

**6.4.3.2 Array Declarators** This section provides the UPC parallel extensions of Section 6.7.5.2 of [2].

**Constraints** When a UPC program is translated in the “dynamic THREADS” environment and the type of the array is shared-qualified but not indefinite layout-qualified, the THREADS lvalue shall occur exactly once in one dimension of the array declarator (including through typedefs). Further, in such cases, the THREADS lvalue shall only occur either alone or when multiplied by a constant expression.

*Semantics*

1. Elements of shared arrays are distributed in round-robin fashion, by chunks of block-size elements, such that the *i*th element has affinity with thread  $(\text{floor}(i/\text{block\_size}) \bmod \text{THREADS})$ .
2. In an array declaration, the type qualifier applies to the elements.
3. For any shared array, *a*, `upc_phaseof (&a)` is zero.

<sup>13</sup>For example, `struct S1 { shared char * p1; };` is legal, whereas `struct S2 { char * shared p2;};` is not.

4. *Example 1:* declarations legal in either static or dynamic translation environments:

```
shared int x[10* THREADS];
shared [ ] int x[10];
```

5. *Example 2:* declarations legal only in a static translation environment:

```
shared int x[10+THREADS];
shared [ ] int x[THREADS];
shared int x[10];
```

6. *Example 3:* declaration of a shared array:

```
shared [3] int x[10];
```

`shared [3]` is the type qualifier of an array, `x`, of 10 integers. `[ 3]` is the layout qualifier.

7. *Example 4:*

```
typedef int S[10];
shared [3] S T[3* THREADS];
```

`shared [3]` applies to the underlying type of `T`, which is `int`, regardless of the typedef. The array is blocked as if it were declared:

```
shared [3] int T[3* THREADS] [10];
```

8. *Example 5:*

```
shared [ ] double D[100];
```

All elements of the array `D` have affinity to thread 0. No `THREADS` dimension is allowed in the declaration of `D`.

```
shared [ ] long *p;
x = p[i];
```

All elements referenced by subscripting or otherwise dereferencing `p` have affinity to the same thread. That thread may be any thread; it does not have to be thread 0.



## 6.5 Statements and Blocks

This section provides the UPC parallel extensions of Section 6.8 of [2].

### Syntax

```
statement:
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  synchronization-statement
```

### 6.5.1 Barrier Statements

#### Syntax

```
synchronization-statement:
  upc_notify expressionopt;
  upc_wait expressionopt;
  upc_barrier expressionopt;
  upc_fence;
```

#### Constraints

1. *expression* shall be an *integer expression*.
2. Each thread shall execute an alternating sequence of `upc_notify` and `upc_wait` statements, starting with a `upc_notify` and ending with a `upc_wait` statement. A synchronization phase consists of the execution of all statements between the completion of one `upc_wait` and the start of the next.

#### Semantics

1. A `upc_wait` statement completes, and the thread enters the next synchronization phase, only after all threads have completed the `upc_notify` statement in the current synchronization phase.<sup>14</sup> `upc_wait` and `upc_notify` are *collective* operations.

<sup>14</sup>Therefore, all threads are entering the same synchronization phase as they complete the `upc_wait` statement.

2. The `upc_fence` statement is equivalent to a *null* strict reference. This ensures that all shared references issued before the fence are complete before any after it are issued.<sup>15</sup>
3. A null strict reference is implied before a `upc_notify` statement and after a `upc_wait` statement.<sup>16</sup>
4. The `upc_wait` statement will generate a run-time error if the value of its expression does not equal the value of the expression by the `upc_notify` statement for the current synchronization phase. No error will be generated if either statement does not have an expression.
5. The `upc_wait` statement will generate a run-time error if the value of its expression differs from any expression on the `upc_wait` and `upc_notify` statements issued by any thread in the current synchronization phase. No error will be generated from a “difference” involving a statement for which no expression is given.
6. The `upc_barrier` statement is equivalent to the compound statement<sup>17</sup>:

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

7. The barrier operations at thread startup and termination have a value of *expression* that is not in the range of user-expressible values.
8. *Example:* The following will result in a run-time error:

```
{upc_notify; upc_barrier; upc_wait;}
```

as it is equivalent to

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

**6.5.2 Iteration Statements** This section provides the UPC parallel extensions of Section 6.8.5 of [2].

### Syntax

1. iteration-statement:
 

```
while (expression) statement
do statement while (expression);
```

<sup>15</sup>One implementation of `upc_fence` may be achieved by a null strict reference: { static shared strict int x; x = x; }.

<sup>16</sup>This implies that shared references executed after the `upc_notify` and before the `upc_wait` may occur in either the synchronization phase containing the `upc_notify` or the next on different threads.

<sup>17</sup>This equivalence is explicit with respect to matching expressions in semantic 6 and collective status in semantic 1.

```

for ( expressionopt; expressionopt; expressionopt) statement
for ( declaration-expressionopt; expressionopt) statement
upc_forall (expressionopt; expressionopt; expressionopt; affinityopt)
statement
2. affinity:
   expressionopt
continue

```

*Constraint* The *expression* for affinity shall have pointer-to-shared type or integer type.

### Semantics

1. `upc_forall` is a *collective* operation in which, for each execution of the loop body, the controlling expression and affinity expression are *single-valued*.<sup>18</sup>
2. The *affinity* field specifies the executions of the loop body that are to be performed by a thread.
3. When *affinity* is of pointer-to-shared type, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value of `upc_threadof (affinity)`. Each iteration of the loop body is executed by precisely one thread.
4. When *affinity* is an integer expression, the loop body of the `upc_forall` statement is executed for each iteration in which the value of `MYTHREAD` equals the value  $affinity \bmod \text{THREADS}$ .
5. When *affinity* is `continue` or not specified, each loop body of the `upc_forall` statement is performed by every thread and semantic 1 does not apply.
6. If the loop body of a `upc_forall` statement contains one or more `upc_forall` statements, either directly or through one or more function calls, the construct is called a “nested `upc_forall`” statement. In a “nested `upc_forall`,” the outermost `upc_forall` statement that has an *affinity* expression which is not `continue` is called the “controlling `upc_forall`” statement. All `upc_forall` statement that are not “controlling” in a “nested `upc_forall`” behave as if their *affinity* expressions were `continue`.
7. If the execution of any loop body of a `upc_forall` statement produces a side effect that affects the execution of another loop body of the same

<sup>18</sup>Note that single-valued implies that all threads agree on the total number of iterations, their sequence, and which threads execute each iteration.

`upc_forall` statement which is executed by a different thread,<sup>19</sup> the behavior is undefined.

8. If any thread terminates or executes a `upc_barrier`, `upc_notify`, or `upc_wait` statement within the dynamic scope of a `upc_forall` statement, the result is undefined. If any thread terminates a `upc_forall` statement using a `break`, `goto`, or `return` statement, the result is undefined. If any thread enters the body of a `upc_forall` statement using a `goto` statement, the result is undefined.<sup>20</sup>
9. *Example: nested upc\_forall:*

```
main () {
    int i, j, k;
    shared float *a, *b, *c;

    upc_forall (i=0; i<N; i++; continue)
        upc_forall (j=0; j<N; j++; &a[j])
            upc_forall (k=0; k<N; k++; &b[k])
                a[j] = b[k] * c[i];
}
```

This example executes all iterations of the `i` and `k` loops on every thread, and executes iterations of the `j` loop on those threads where `upc_threadof (&a[j])` equals the value of `MYTHREAD`.

## 6.6 Preprocessing Directives

This section provides the UPC parallel extensions of Section 6.10 of [2].

### 6.6.1 UPC Pragmas

#### *Semantics*

1. If the preprocessing token `upc` follows the `pragma` immediately, no macro replacement is performed and the directive shall have one of the following forms:

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

<sup>19</sup>This semantic implies that side effects on the same thread have defined behavior, just as in the `for` statement.

<sup>20</sup>The `continue` statement behaves as defined in Section 6.8.6.2 of [2]; equivalent to a `goto` the end of the loop body.

2. These pragmas affect the strict or relaxed categorization of references to shared objects where the referenced type is neither strict-qualified nor relaxed-qualified. Such references shall be strict if a strict pragma is in effect, or relaxed if a relaxed pragma is in effect.
3. Shared references that are not categorized by either referenced type or by these pragmas behave in an implementation-defined manner in which either all such references are strict or all are relaxed. Users wishing portable programs are strongly encouraged to categorize all shared references either by using type qualifiers, these directives, or by including `upc_strict.h` or `upc_relaxed.h`.
4. The pragmas shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When they are outside external declarations, they apply until another such pragma or the end of the translation unit. When inside a compound statement, they apply until the end of the compound statement; at the end of the compound statement the state of the pragmas is restored to that preceding the compound statement. If these pragmas are used in any other context, their behavior is undefined.

### 6.6.2 *Predefined Macro Names*

1. The following macro name shall be defined by the implementation:
  - `_UPC_`: the integer constant 1, indicating a conforming implementation
  - `_UPC_VERSION_`: the integer constant 200310L
2. The following macro names are conditionally defined by the implementation:
  - `_UPC_DYNAMIC_THREADS_`: the integer constant 1 in the dynamic THREADS translation environment; otherwise, undefined
  - `_UPC_STATIC_THREADS_`: the integer constant 1 in the static THREADS translation environment; otherwise, undefined

## 7 LIBRARY

### 7.1 Standard Headers

The section provides the UPC parallel extensions of Section 7.1.2 of [2].

1. The standard headers are

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

2. `upc_strict.h` shall contain at least

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

3. `upc_relaxed.h` shall contain at least

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

## 7.2 UPC Utilities <upc.h>

This section provides the UPC parallel extensions of Section 7.20 of [2]. All of the characteristics of library functions described in Section 7.1.4 of [2] apply to these as well.

### 7.2.1 Termination of All Threads

#### *Synopsis*

```
upc_global_exit(int status)
```

*Description* `upc_global_exit()` flushes all I/O, releases all storage, and terminates the execution for all active threads.

**7.2.2 Shared Memory Allocation Functions** The UPC memory allocation functions return, if successful, a pointer-to-shared which is suitably aligned so that it may be assigned to a pointer-to-shared of any type. The pointer has zero phase and points to the start of the space allocated. If the space cannot be allocated, a null pointer-to-shared is returned.

#### 7.2.2.1 `upc_global_alloc` Function

#### *Synopsis*

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

```
shared void *upc_global_alloc(size_t nblocks, size_t nbytes);
    nblocks : number of blocks
    nbytes  : block size
```

#### *Description*

1. The `upc_global_alloc` allocates shared space compatible with the declaration

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

2. The `upc_global_alloc` function is not a *collective* function. If called by multiple threads, all threads that make the call get different allocations. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.

### 7.2.2.2 `upc_all_alloc` Function

#### *Synopsis*

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

```
shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
    nblocks : number of blocks
    nbytes  : block size
```

#### *Description*

1. `upc_all_alloc` is a *collective* function with *single-valued* arguments.
2. The `upc_all_alloc` function allocates shared space compatible with the following declaration:

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

3. The `upc_all_alloc` function returns the same pointer value on all threads. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.
4. The dynamic lifetime of an allocated object extends from the time that any thread completes the call to `upc_all_alloc` until any thread has deallocated the object.

### 7.2.2.3 `upc_alloc` Function

#### *Synopsis*

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

```
shared void *upc_alloc(size_t nbytes);
    nbytes : total number of bytes to allocate
```

#### *Description*

1. The `upc_alloc` function allocates shared space of at least `nbytes` bytes with affinity to the calling thread.

2. `upc_alloc` is similar to `malloc()` except that it returns a pointer-to-shared value. It is not a *collective* function. If `nbytes` is zero, the result is a null pointer-to-shared.

#### 7.2.2.4 `upc_local_alloc` Function *Deprecated*

##### *Synopsis*

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

```
shared void*upc_local_alloc(size_t nblocks, size_t nbytes);
nblocks : number of blocks
nbytes : block size
```

##### *Description*

1. The `upc_local_alloc` function is deprecated and should not be used. UPC programs should use the `upc_alloc` function instead. Support may be removed in future versions of this specification.
2. The `upc_local_alloc` function allocates shared space of at least `nblocks * nbytes` bytes with affinity to the calling thread. If `nblocks*nbytes` is zero, the result is a null pointer-to-shared.
3. `upc_local_alloc` is similar to `malloc()` except that it returns a pointer-to-shared value. It is not a *collective* function.

#### 7.2.2.5 `upc_free` Function

##### *Synopsis*

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

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

**Description** The `upc_free` function frees the dynamically allocated shared storage pointed to by `ptr`. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `upc_alloc`, `upc_global_alloc`, `upc_all_alloc`, or `upc_local_alloc`, function, or if the space has been deallocated by a previous call, by any thread,<sup>21</sup> to `upc_free`, the behavior is undefined.

<sup>21</sup> That is, only one thread may call `upc_free` for each allocation.



## 7.2.3 Pointer-to-Shared Manipulation Functions

### 7.2.3.1 upc\_threadof Function

#### *Synopsis*

```
#include <upc.h>

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

*Description* The `upc_threadof` function returns the number of the thread that has affinity to the shared object pointed to by `ptr`.

### 7.2.3.2 upc\_phaseof Function

#### *Synopsis*

```
#include <upc.h>

size_t upc_phaseof (shared void *ptr);
```

*Description* The `upc_phaseof` function returns the phase component of the pointer-to-shared argument.

### 7.2.3.3 upc\_resetphase Function

#### *Synopsis*

```
#include <upc.h>
shared void *upc_resetphase (shared void *ptr);
```

*Description* The `upc_resetphase` function returns a pointer-to-shared which is identical to its input except that it has zero phase.

### 7.2.3.4 upc\_addrfield Function

#### *Synopsis*

```
#include <upc.h>

size_t upc_addrfield (shared void *ptr);
```

*Description* The `upc_addrfield` function returns an implementation-defined value reflecting the “local address” of the object pointed to by the pointer-to-shared argument.

### 7.2.3.5 upc\_affinitysize Function

#### *Synopsis*

```
#include <upc.h>
size_t upc_affinitysize (size_t totalsize, size_t nbytes,
                        size_t threadid);
totalsize: the total size of the allocation in bytes
nbytes: the number of bytes in a block
threadid: the thread whose affinitysize is to be evaluated
```

#### *Description*

1. `upc_affinitysize` is a convenience function that calculates the exact size of the local portion of the data in a shared object with affinity to a given thread.
2. In the case of a dynamically allocated shared object, the `totalsize` argument shall be `nbytes*nblocks` and the `nbytes` argument shall be `nbytes`, where `nblocks` and `nbytes` are exactly as passed to `upc_global_alloc` or `upc_all_alloc` when the object was allocated.
3. In the case of a statically allocated shared object with declaration

```
shared[b] t d[s];
```

the `totalsize` argument shall be `s * sizeof (t)` and the `nbytes` argument shall be `b * sizeof (t)`. If block size is unspecified, `nbytes` shall be 1. If the block size is indefinite, `nbytes` shall be 0.

4. `threadid` shall be a value in `0..(THREADS-1)`.

## 7.2.4 Lock Functions

### 7.2.4.1 Type

1. The type declared is

```
upc_lock_t
```

2. The type `upc_lock_t` is an opaque UPC type. `upc_lock_t` is a shared data type with incomplete type (as defined in Section 6.2.5) of [2]. Objects of type `upc_lock_t` may therefore only be manipulated through pointers.

### 7.2.4.2 `upc_global_lock_alloc` Function

#### *Synopsis*

```
#include <upc.h>

upc_lock_t *upc_global_lock_alloc (void);
```

#### *Description*

1. The `upc_global_lock_alloc` function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.
2. The `upc_global_lock_alloc` function is not a *collective* function. If called by multiple threads, all threads that make the call get different allocations.

### 7.2.4.3 `upc_all_lock_alloc` Function

#### *Synopsis*

```
#include <upc.h>

upc_lock_t *upc_all_lock_alloc (void);
```

#### *Description*

1. The `upc_all_lock_alloc` function dynamically allocates a lock and returns a pointer to it. The lock is created in an unlocked state.
2. The `upc_all_lock_alloc` is a *collective* function. The return value on every thread points to the same lock object.

### 7.2.4.4 `upc_lock_free` Function

#### *Synopsis*

```
#include <upc.h>

void upc_lock_free (upc_lock_t *ptr);
```

#### *Description*

1. The `upc_lock_free` function frees all resources associated with the dynamically allocated `upc_lock_t` pointed to by `ptr`. If `ptr` is a null

pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `upc_global_lock_alloc` or `upc_all_lock_alloc` function, or if the lock has been deallocated by a previous call to `upc_lock_free`, the behavior is undefined.

2. `upc_lock_free` succeeds regardless of whether the referenced lock is currently unlocked or currently locked (by any thread).
3. Any subsequent calls to locking functions from any threads using `ptr` have undefined effects.

#### 7.2.4.5 `upc_lock` Function

##### *Synopsis*

```
#include <upc.h>

void upc_lock(upc_lock_t *ptr);
```

##### *Description*

1. The `upc_lock` function locks a shared variable, of type `upc_lock_t`, pointed to by the pointer given as argument.
2. If the lock is not used by another thread, the thread making the call gets the lock and the function returns. Otherwise, the function keeps trying to get access to the lock.
3. A null strict reference is implied after a call to `upc_lock()`.
4. If the calling thread is already holding the lock referenced by `ptr` [i.e., it has locked it previously using `upc_lock()` or `upc_lock_attempt()`, but not unlocked it], the result is undefined.

#### 7.2.4.6 `upc_lock_attempt` Function

##### *Synopsis*

```
#include <upc.h>

int upc_lock_attempt(upc_lock_t *ptr);
```

##### *Description*

1. The `upc_lock_attempt` function tries to lock a shared variable, of type `upc_lock_t`, pointed to by the pointer given as argument.
2. If the lock is not used by another thread, the thread making the call gets the lock and the function returns 1. Otherwise, the function returns 0.

3. A null strict reference is implied after a call to `upc_lock_attempt()` that returns 1.
4. If the calling thread is already holding the lock referenced by `ptr` [i.e., it has locked it previously using `upc_lock()` or `upc_lock_attempt()`, but not unlocked it], the result is undefined.

#### 7.2.4.7 `upc_unlock` Function

##### *Synopsis*

```
#include <upc.h>

void upc_unlock(upc_lock_t *ptr);
```

##### *Description*

1. The `upc_unlock` function frees the lock and does not return any value.
2. A null strict reference is implied before a call to `upc_unlock()`.

### 7.2.5 *Shared String-Handling Functions*

#### 7.2.5.1 `upc_memcpy` Function

##### *Synopsis*

```
#include <upc.h>

void upc_memcpy(shared void *dst, shared const void *src,
                size_t n);
```

##### *Description*

1. The `upc_memcpy` function copies `n` characters from a shared object having affinity with one thread to a shared object having affinity with the same or another thread. If copying takes place between objects that overlap, the behavior is undefined.
2. The `upc_memcpy` function treats the `dst` and `src` pointers as if they had type

```
shared[] char[n]
```

The effect is equivalent to copying the entire contents from one shared array object with this type (the `src` array) to another shared array object with this type (the `dst` array).

### 7.2.5.2 `upc_memget` Function

#### *Synopsis*

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

```
void upc_memget(void *dst, shared const void *src, size_t n);
```

#### *Description*

1. The `upc_memget` function copies `n` characters from a shared object with affinity to any single thread to a private object on the calling thread. If copying takes place between objects that overlap, the behavior is undefined.
2. The `upc_memget` function treats the `src` pointer as if it had type

```
shared[] char [n]
```

The effect is equivalent to copying the entire contents from one shared array object with this type (the `src` array) to a private array object (the `dst` array) declared with the type

```
char [n]
```

### 7.2.5.3 `upc_memput` Function

#### *Synopsis*

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

```
void upc_memput(shared void *dst, const void *src, size_t n);
```

#### *Description*

1. The `upc_memput` function copies `n` characters from a private object on the calling thread to a shared object with affinity to any single thread. If copying takes place between objects that overlap, the behavior is undefined.
2. The `upc_memput` function is equivalent to copying the entire contents from a private array object (the `src` array) declared with the type

```
char [n]
```

to a shared array object (the `dst` array) with the type

```
shared[] char[n]
```

#### 7.2.5.4 `upc_memset` Function

##### *Synopsis*

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

```
void upc_memset(shared void *dst, int c, size_t n);
```

##### *Description*

1. The `upc_memset` function copies the value of `c`, converted to an unsigned char, to a shared object with affinity to any single thread. The number of bytes set is `n`.
2. The `upc_memset` function treats the `dst` pointer as if had type

```
shared[] char[n]
```

The effect is equivalent to setting the entire contents of a shared array object with this type (the `dst` array) to the value `c`.

## ACKNOWLEDGMENTS

Many scientists have contributed to the ideas and concepts behind these specifications. They are too many to mention here, but we would like to cite the contributions of David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren, who have contributed to the initial UPC language concepts and specifications. We also would like to acknowledge the role of the participants in the first UPC workshop, held in May 2000 in Bowie, Maryland, in which the specifications of this version were discussed. In particular, we would like to acknowledge the support and participation of Compaq, Cray, HP, Sun, and CSC. We would like also to acknowledge the abundant input of Kevin Harris and Sébastien Chauvin and the efforts of Lauren Smith. The work by Brian Wibecan and Greg Fischer was invaluable in bringing these specifications to version 1.0.

Version 1.1 is the result of the contributions of many in the UPC community: most important, participants in the second UPC workshop held in March 2002 in Washington, DC. In addition to the continued support of all

those mentioned above, the efforts of Dan Bonachea were invaluable in this effort.

## REFERENCES

1. W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and Language Specification*, CCS-TR-99-157, IDA/CCS, Bowie, MD, May 1999.
2. *Programming Languages—C*, ISO/IEC 9899:1999(E), ISO/IEC, May 2000.





# UPC Collective Operations Specifications, v1.0

Steven Seidel, David Greenberg, and Elizabeth Wiebel

## Introduction

### 1 Scope

### 2 Definitions

### 3 Common Requirements

### 4 Collectives Library

#### 4.1 Standard Header

#### 4.2 Relocalization Operations

4.2.1 `upc_all_broadcast` Function

4.2.2 `upc_all_scatter` Function

4.2.3 `upc_all_gather` Function

4.2.4 `upc_all_gather_all` Function

4.2.5 `upc_all_exchange` Function

4.2.6 `upc_all_permute` Function

#### 4.3 Computational Operations

4.3.1 Computational Operations

4.3.2 `upc_all_reduce` Function

4.3.3 `upc_all_prefix_reduce` Function

4.3.4 `upc_all_sort` Function

## Acknowledgments

## References

---

*UPC: Distributed Shared Memory Programming*, by Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick  
Published 2005 John Wiley & Sons, Inc.

## INTRODUCTION

The earliest version of this specification was authored by Elizabeth Wiebel and David Greenberg and appeared in March 2002 as CCS-TR-02-159 [1]. UPC collectives have been discussed at workshops at SC2001, SC2002, SC2003, and at workshops held at George Washington University in March 2002 and May 2003.

### 1 SCOPE

1. This document describes UPC functions that supplement UPC. All UPC specifications from v1.1.1 [2] are considered part of this specification, and therefore will not be addressed in this document.
2. Small parts of UPC v1.1.1 may be repeated for self-containment and clarity of the functions defined here.

### 2 DEFINITIONS

1. **Collective:** a requirement placed on some language operations that constrains invocations of such operations to be matched<sup>1</sup> across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.
2. **Single-valued:** an operand to a collective operation, which has the same value on every thread. The behavior of the operation is otherwise undefined.
3. All but one of the functions defined in this specification include a figure that roughly illustrates how blocks of data are copied from one thread to another. Four threads labeled  $T_0$ ,  $T_1$ ,  $T_2$ , and  $T_3$  are shown in each figure along with a suitable number of blocks of data labeled  $D_i$ . These figures are not intended to represent the full generality of the associated functions. The figures do not correspond to the example code segments unless otherwise noted. The figures and the example code segments should not be viewed as formal parts of this specification.

### 3 COMMON REQUIREMENTS

1. The following requirements apply to all of the functions defined in this document whose names begin `upc_all_...`
2. All of the functions are collective.
3. All collective function arguments are single-valued.

<sup>1</sup>A collective operation need not provide any actual synchronization between threads unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely on such unspecified synchronization for correctness.

4. Collective functions may not be called between `upc_notify` and the corresponding `upc_wait`.
5. The last argument of each collective function is the variable `sync_mode` of type `upc_flag_t`. Values of `sync_mode` are formed by or-ing together a constant of the form `UPC_IN_XSYNC` and a constant of the form `UPC_OUT_YSYNC`, where X and Y may be NO, MY, or ALL. If `sync_mode` has the value `(UPC_IN_XSYNC | UPC_OUT_YSYNC)`, then if X is
  - NO the collective function may begin to read or write data when the first thread has entered the collective function call
  - MY the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call
  - ALL the collective function may begin to read or write data only after all threads have entered the collective function call<sup>2</sup>

and if Y is

- NO the collective function may read and write data until the last thread has returned from the collective function call
- MY the collective function call may return in a thread only after all reads and writes of data with affinity to the thread are complete<sup>3</sup>
- ALL the collective function call may return only after all reads and writes of data are complete<sup>4</sup>

`UPC_IN_XSYNC` alone is equivalent to `(UPC_IN_XSYNC | UPC_OUT_ALLSYNC)`, `UPC_OUT_XSYNC` alone is equivalent to `(UPC_IN_ALLSYNC | UPC_OUT_XSYNC)`, and 0 is equivalent to `(UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC)`, where X is NO, MY, or ALL.

*Forward reference:* `upc_flag_t` (4.1.2).

## 4 COLLECTIVES LIBRARY

### 4.1 Standard Header

1. The standard header is

```
<upc_collective.h>
```

<sup>2</sup>`UPC_IN_ALLSYNC` requires the collective function to guarantee that after all threads have entered the collective function call, all threads will read the same values of the input data.

<sup>3</sup>`UPC_OUT_MYSYNC` requires the collective function to guarantee that after a thread returns from the collective function call, the thread will not read any earlier values of the output data with affinity to that thread.

<sup>4</sup>`UPC_OUT_ALLSYNC` requires the collective function to guarantee that after a thread returns from the collective function call, the thread will not read any earlier values of the output data. `UPC_OUT_ALLSYNC` is not required to provide an “implied” barrier. For example, if the entire collective operation has been completed by a certain thread before some other threads have reached their corresponding function calls, that thread may exit its call.

2. `upc_collective.h` contains definitions of the types `upc_flag_t` and `upc_op_t`. `upc_flag_t` and `upc_op_t` are of integral type.

*Forward reference:* `upc_op_t` (4.3.1).

## 4.2 Relocalization Operations

### 4.2.1 `upc_all_broadcast` Function

#### Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_broadcast(shared void*dst, shared const void*src,
    size_t nbytes, upc_flag_t sync_mode);
nbytes: the number of bytes in a block
```

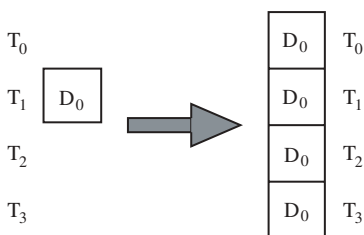
#### Description

1. The `upc_all_broadcast` function (Figure 1) copies a block of memory with affinity to a single thread to a block of shared memory on each thread. The number of bytes in each block is `nbytes`. If copying takes place between objects that overlap, the behavior is undefined.
2. `nbytes` must be strictly greater than zero.
3. The `upc_all_broadcast` function treats the `src` pointer as if it pointed to a shared memory area with the type

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

The effect is equivalent to copying the entire array pointed to by `src` to each block of `nbytes` bytes of a shared array `dst` with the type

```
shared[nbytes] char[nbytes* THREADS]
```



**Figure 1** `upc_all_broadcast`

4. The target of the `dst` pointer must have affinity to thread 0.
5. The `dst` pointer is treated as if it has phase 0.

### Examples

1. This example corresponds to Figure 1:

```
shared int A [THREADS];
shared int B [THREADS];
//Initialize A.
upc_barrier;
upc_all_broadcast (B, &A[1] , sizeof(int),
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

2. Example of `upc_all_broadcast`:

```
#define NELEMS 10
shared [ ] int A [NELEMS];
shared [NELEMS] int B [NELEMS* THREADS];
//Initialize A.
upc_all_broadcast (B, A, sizeof(int)*NELEMS,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

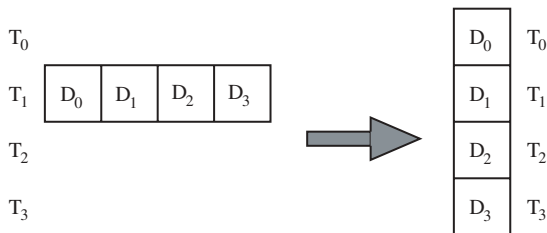
3. In this example,  $(A[3], A[4])$  is broadcast to  $(B[0], B[1])$ ,  $(B[10], B[11])$ ,  $(B[20], B[21])$ , ...,  $(B[NELEMS * (THREADS-1)], B[NELEMS * (THREADS-1) + 1])$ :

```
#define NELEMS 10
shared [NELEMS] int A [NELEMS* THREADS];
shared [NELEMS] int B [NELEMS* THREADS];
//Initialize A.
upc_barrier;
upc_all_broadcast (B, &A [3] , sizeof(int)*2,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

#### 4.2.2 `upc_all_scatter` Function

##### Synopsis

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



**Figure 2** upc\_all\_scatter

```
void upc_all_scatter(shared void *dst, shared const void *src,
    size_t nbytes, upc_flag_t sync_mode);
nbytes: the number of bytes in a block
```

*Description*

1. The upc\_all\_scatter function (Figure 2) copies the *i*th block of an area of shared memory with affinity to a single thread to a block of shared memory with affinity to the *i*th thread. The number of bytes in each block is *nbytes*. If copying takes place between objects that overlap, the behavior is undefined.
2. *nbytes* must be strictly greater than zero.
3. The upc\_all\_scatter function treats the *src* pointer as if it pointed to a shared memory area with the type

```
shared[] char[nbytes* THREADS]
```

and it treats the *dst* pointer as if it pointed to a shared memory area with the type

```
shared[nbytes] char[nbytes* THREADS]
```

4. The target of the *dst* pointer must have affinity to thread 0.
5. The *dst* pointer is treated as if it has phase 0.
6. For each thread *i*, the effect is equivalent to copying the *i*th block of *nbytes* bytes pointed to by *src* to the block of *nbytes* bytes pointed to by *dst* that has affinity to thread *i*.

*Examples*

1. Example of upc\_all\_scatter for the dynamic threads compilation environment (this example corresponds to Figure 2):

```

#define NUMELEMS 10
#define SRC_THREAD 1
shared int *A;
shared[] int *myA, *srcA;
shared[NUMELEMS] int B [NUMELEMS* THREADS];

// allocate and initialize an array distributed across all threads
A = upc_all_alloc(THREADS, THREADS* NUMELEMS* sizeof(int));
myA = (shared[] int *) &A[MYTHREAD];
for (i=0; i<NUMELEMS* THREADS; i++)
    myA[i] = i + NUMELEMS* THREADS* MYTHREAD; // (for example)
// scatter the SRC_THREAD' s row of the array
srcA=(shared[] int *) &A[SRC_THREAD];
upc_barrier;
upc_all_scatter(B, srcA, sizeof(int)* NUMELEMS,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;

```

2. Example of `upc_all_scatter` for the static threads compilation environment:

```

#define NELEMS 10
shared[] int A [NELEMS* THREADS];
shared[NELEMS] int B [NELEMS* THREADS];
// Initialize A.
upc_all_scatter(B, A, sizeof(int)* NELEMS,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);

```

### 4.2.3 `upc_all_gather` Function

#### *Synopsis*

```

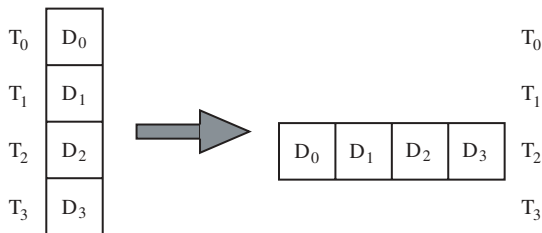
#include <upc.h>
#include <upc_collective.h>
void upc_all_gather(shared void *dst, shared const void *src,
    size_t nbytes, upc_flag_t sync_mode);
nbytes: the number of bytes in a block

```

#### *Description*

1. The `upc_all_gather` function (Figure 3) copies a block of shared memory that has affinity to the  $i$ th thread to the  $i$ th block of a shared memory area





**Figure 3** upc\_all\_gather

that has affinity to a single thread. The number of bytes in each block is `nbytes`. If copying takes place between objects that overlap, the behavior is undefined.

2. `nbytes` must be strictly greater than zero.
3. The `upc_all_gather` function treats the `src` pointer as if it pointed to a shared memory area of `nbytes` bytes on each thread and therefore had type

```
shared[nbytes] char[nbytes* THREADS]
```

and it treats the `dst` pointer as if it pointed to a shared memory area with the type

```
shared[] char[nbytes* THREADS]
```

4. The target of the `src` pointer must have affinity to thread 0.
5. The `src` pointer is treated as if it has phase 0.
6. For each thread `i`, the effect is equivalent to copying the block of `nbytes` bytes pointed to by `src` that has affinity to thread `i` to the `i`th block of `nbytes` bytes pointed to by `dst`.

### Examples

1. Example of `upc_all_gather` for the static threads compilation environment:

```
#define NELEMS 10
shared[NELEMS] int A[NELEMS* THREADS];
shared[] int B[NELEMS* THREADS];
// Initialize A.
upc_all_gather(B, A, sizeof(int)*NELEMS,
              UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

2. Example of `upc_all_gather` for the dynamic threads compilation environment:

```
#define NELEMS 10
shared[NELEMS] int A[NELEMS* THREADS];
shared[] int *B;
B = (shared[] int*) upc_all_alloc(1, NELEMS* THREADS* sizeof(int));
// Initialize A.
upc_barrier;
upc_all_gather(B, A, sizeof(int)*NELEMS,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

#### 4.2.4 `upc_all_gather_all` Function

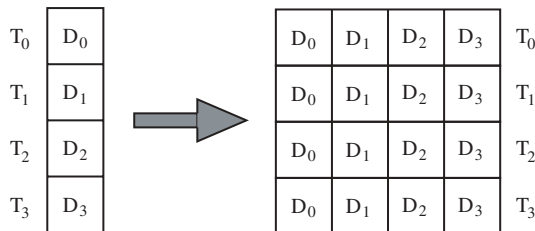
##### Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_gather_all(shared void *dst, shared const void *src,
    size_t nbytes, upc_flag_t sync_mode);
nbytes: the number of bytes in a block
```

##### Description

1. The `upc_all_gather_all` function (Figure 4) copies a block of memory from one shared memory area with affinity to the  $i$ th thread to the  $i$ th block of a shared memory area on each thread. The number of bytes in each block is `nbytes`. If copying takes place between objects that overlap, the behavior is undefined.
2. `nbytes` must be strictly greater than zero.
3. The `upc_all_gather_all` function treats the `src` pointer as if it pointed to a shared memory area of `nbytes` bytes on each thread and therefore had type

```
shared[nbytes] char [nbytes* THREADS]
```



**Figure 4** `upc_all_gather_all`

and it treats the `dst` pointer as if it pointed to a shared memory area with the type

```
shared [nbytes* THREADS] char [nbytes* THREADS* THREADS]
```

4. The targets of the `src` and `dst` pointers must have affinity to thread 0.
5. The `src` and `dst` pointers are treated as if they have phase 0.
6. The effect is equivalent to copying the `i`th block of `nbytes` bytes pointed to by `src` to the `i`th block of `nbytes` bytes pointed to by `dst` that has affinity to each thread.

### Examples

1. Example of `upc_all_gather_all` for the static threads compilation environment:

```
#define NELEMS 10
shared [NELEMS] int A [NELEMS* THREADS];
shared [NELEMS* THREADS] int B [THREADS] [NELEMS* THREADS];
// Initialize A.
upc_barrier;
upc_all_gather_all (B, A, sizeof (int)*NELEMS,
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

2. Example of `upc_all_gather_all` for the dynamic threads compilation environment:

```
#define NELEMS 10
shared [NELEMS] int A [NELEMS* THREADS];
shared int *Bdata;
shared [] int *myB;

Bdata = upc_all_alloc (THREADS* THREADS, NELEMS* sizeof (int));
myB = (shared [] int *) &Bdata[MYTHREAD];

// Bdata contains THREADS* THREADS* NELEMS elements.
// myB is MYTHREAD' s row of Bdata.
// Initialize A.
upc_all_gather_all (Bdata, A, NELEMS* sizeof (int),
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

## 4.2.5 upc\_all\_exchange Function

### Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_exchange(shared void*dst, shared const void*src,
    size_t nbytes, upc_flag_t sync_mode);
nbytes: the number of bytes in a block
```

### Description

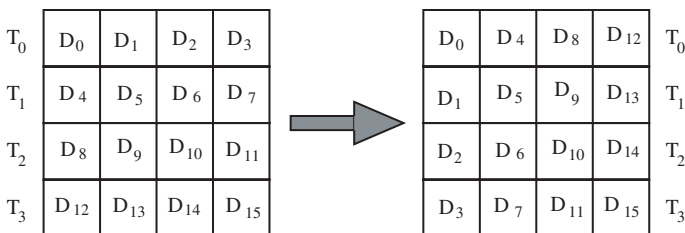
1. The `upc_all_exchange` function (Figure 5) copies the  $i$ th block of memory from a shared memory area that has affinity to thread  $j$  to the  $j$ th block of a shared memory area that has affinity to thread  $i$ . The number of bytes in each block is `nbytes`. If copying takes place between objects that overlap, the behavior is undefined.
2. `nbytes` must be strictly greater than 0.
3. The `upc_all_exchange` function treats the `src` pointer and the `dst` pointer as if each pointed to a shared memory area of `nbytes*THREADS` bytes on each thread and therefore had type

```
shared [nbytes*THREADS] char[nbytes*THREADS*THREADS]
```

4. The targets of the `src` and `dst` pointers must have affinity to thread 0.
5. The `src` and `dst` pointers are treated as if they have phase 0.
6. For each pair of threads  $i$  and  $j$ , the effect is equivalent to copying the  $i$ th block of `nbytes` bytes that has affinity to thread  $j$  pointed to by `src` to the  $j$ th block of `nbytes` bytes that has affinity to thread  $i$  pointed to by `dst`.

### Examples

1. Example of `upc_all_exchange` for the static threads compilation environment:



**Figure 5** `upc_all_exchange`

```

#define NELEMS 10
shared [NELEMS* THREADS] int A [THREADS] [NELEMS* THREADS];
shared [NELEMS* THREADS] int B [THREADS] [NELEMS* THREADS];
// Initialize A.
upc_barrier;
upc_all_exchange (B, A, NELEMS* sizeof (int),
    UPC_IN_NOSYNC | UPC_OUT_NOSYNC);

upc_barrier;

```

## 2. Example of upc\_all\_exchange for the dynamic threads compilation environment:

```

#define NELEMS 10
shared int *Adata, *Bdata;
shared [] int *myA, *myB;
int i;

Adata = upc_all_alloc (THREADS* THREADS, NELEMS* sizeof (int));
myA = (shared [] int *) &Adata [MYTHREAD];
Bdata = upc_all_alloc (THREADS* THREADS, NELEMS* sizeof (int));
myB = (shared [] int *) &Bdata [MYTHREAD];

// Adata and Bdata contain THREADS* THREADS* NELEMS elements.
// myA and myB are MYTHREAD' s rows of Adata and Bdata, resp.

// Initialize MYTHREAD' s row of A. For example,
for (i=0; i<NELEMS* THREADS; i++)
    myA [i] = MYTHREAD* 10 + i;

upc_all_exchange ( Bdata, Adata, NELEMS* sizeof (int),
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);

```

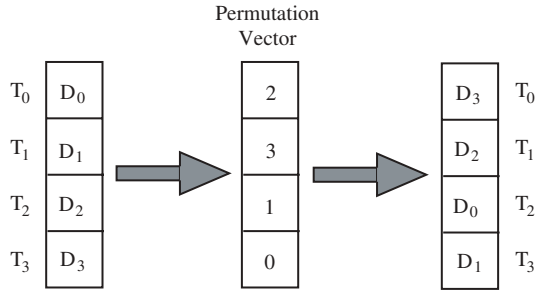
### 4.2.6 upc\_all\_permute Function

#### *Synopsis*

```

#include <upc.h>
#include <upc_collective.h>
void upc_all_permute (shared void *dst, shared const void *src,
    shared const int *perm, size_t nbytes,
    upc_flag_t sync_mode);
nbytes: the number of bytes in a block

```



**Figure 6** upc\_all\_permute

### Description

1. The `upc_all_permute` function (Figure 6) copies a block of memory from a shared memory area that has affinity to the `i`th thread to a block of a shared memory that has affinity to thread `perm[i]`. The number of bytes in each block is `nbytes`. If copying takes place between objects that overlap, the behavior is undefined.
2. `nbytes` must be strictly greater than zero.
3. `perm[0..THREADS-1]` must contain `THREADS` distinct values: `0, 1, ..., THREADS-1`.
4. The `upc_all_permute` function treats the `src` pointer and the `dst` pointer as if each pointed to a shared memory area of `nbytes` bytes on each thread and therefore had type

```
shared[nbytes] char [nbytes* THREADS]
```

5. The targets of the `src`, `perm`, and `dst` pointers must have affinity to thread 0.
6. The `src` and `dst` pointers are treated as if they have phase 0.
7. The effect is equivalent to copying the block of `nbytes` bytes that has affinity to thread `i` pointed to by `src` to the block of `nbytes` bytes that has affinity to thread `perm[i]` pointed to by `dst`.

**Example:** `upc_all_permute`:

```
#define NELEMS 10
shared[NELEMS] int A[NELEMS* THREADS], B[NELEMS* THREADS];
shared int P[THREADS];
// Initialize A and P.
upc_barrier;
upc_all_permute(B, A, P, sizeof(int)*NELEMS,
```

```
UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

## 4.3 Computational Operations

### 4.3.1 Computational Operations

1. A variable of type `upc_op_t` can have the following values:
  - `UPC_ADD`: addition.
  - `UPC_MULT`: multiplication.
  - `UPC_AND`: bitwise AND for integer and character variables. Results are undefined for floating-point numbers.
  - `UPC_OR`: bitwise OR for integer and character variables. Results are undefined for floating-point numbers.
  - `UPC_XOR`: bitwise XOR for integer and character variables. Results are undefined for floating-point numbers.
  - `UPC_LOGAND`: logical AND for all variable types.
  - `UPC_LOGOR`: logical OR for all variable types.
  - `UPC_MIN`: for all data types, finds the minimum value.
  - `UPC_MAX`: for all data types, finds the maximum value.
  - `UPC_FUNC`: uses the specified commutative function `func` to operate on the data in the `src` array at each step.
  - `UPC_NONCOMM_FUNC`: uses the specified noncommutative function `func` to operate on the data in the `src` array at each step.
2. The operations represented by a variable of type `upc_op_t` (including user-provided operators) are assumed to be associative. A reduction or a prefix reduction whose result is dependent on the order of operator evaluation will have undefined results.<sup>5</sup>

*Forward references:* reduction (4.3.2), prefix reduction (4.3.3).

3. The operations represented by a variable of type `upc_op_t` (except those provided using `UPC_NONCOMM_FUNC`) are assumed to be commutative. A reduction or a prefix reduction (using operators other than `UPC_NONCOMM_FUNC`) whose result is dependent on the order of the operands will have undefined results.

*Forward references:* reduction (4.3.2), prefix reduction (4.3.3).

<sup>5</sup> Implementations are not obligated to prevent failures that might arise because of a lack of associativity of built-in functions due to floating-point roundoff or overflow.

### 4.3.2 upc\_all\_reduce Function

#### Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_reduceT(shared void *dst, shared const void *src,
    upc_op_t op, size_t nelems, size_t blk_size,
    TYPE (*func) (TYPE, TYPE), upc_flag_t sync_mode);
nelems: the number of elements
blk_size: the number of elements in a block
```

#### Description

1. The function prototype above represents the 11 variations of the `upc_all_reduceT` function where `T` and `TYPE` have the following correspondences:

T	TYPE	T	TYPE
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

For example, if `T` is `C`, `TYPE` must be `signed char`.

2. If the value of `blk_size` passed to `upc_all_reduceT` is greater than zero `upc_all_reduceT` treats the `src` pointer as if it pointed to a shared memory area of `nelems` elements of type `TYPE` and blocking factor `blk_size`, and therefore had type

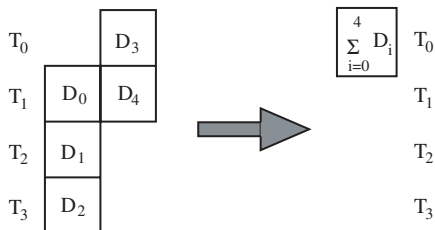
```
shared[blk_size] TYPE[nelems]
```

3. If the value of `blk_size` passed to `upc_all_reduceT` is zero, `upc_all_reduceT` treats the `src` pointer as if it pointed to a shared memory area of `nelems` elements of type `TYPE` with an indefinite layout qualifier, and therefore had type<sup>6</sup>

```
shared[] TYPE[nelems]
```

<sup>6</sup>Note the `upc_blocksize(src)==0` if `src` has this type, so the argument value 0 has a natural connection to the block size of `src`.





**Figure 7** upc\_all\_reduce with Addition Operator

4. The phase of the `src` pointer is respected when referencing array elements, as specified in items 2 and 3.
5. `upc_all_reduceT` treats the `dst` pointer as having type

`shared TYPE *`

and at function exit the value of the `TYPE` shared object referenced by `dst` is

`src[0] ⊕ src[1] ⊕ ⋯ ⊕ src[nelems-1]`

where “ $\oplus$ ” is the operator specified by the variable `op`.

**Example:** upc\_all\_reduce of type `long` UPC\_ADD (Figure 7)

```
#define BLK_SIZE 3
#define NELEMS 10
shared [BLK_SIZE] long A [NELEMS* THREADS];
shared long *B;
long result;
// Initialize A. The result below is defined only on thread 0.
upc_barrier;
upc_all_reduceL(B, A, UPC_ADD, NELEMS* THREADS, BLK_SIZE,
  NULL, UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

### 4.3.3 upc\_all\_prefix\_reduce Function

#### Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_prefix_reduceT(shared void *dst, shared const void *src
  upc_op_t op, size_t nelems, size_t blk_size,
  TYPE (*func) (TYPE, TYPE),
```

```

upc_flag_t sync_mode);
nelems: the number of elements
blk_size: the number of elements in a block

```

### Description

1. The function prototype above represents the **11** variations of the `upc_all_reduceT`, function where `T` and `TYPE` have the following correspondences:

T	TYPE	T	TYPE
C	signed char	L	signed long
UC	unsigned char	UL	unsigned long
S	signed short	F	float
US	unsigned short	D	double
I	signed int	LD	long double
UI	unsigned int		

For example, if `T` is `C`, then `TYPE` must be `signed char`.

2. If the value of `blk_size` passed to `upc_all_prefix_reduceT` is greater than zero, `upc_all_prefix_reduceT` treats the `src` pointer and the `dst` pointer as if each pointed to a shared memory area of `nelems` elements of type `TYPE` and blocking factor `blk_size`, and therefore had type

```
shared[blk_size] TYPE[nelems]
```

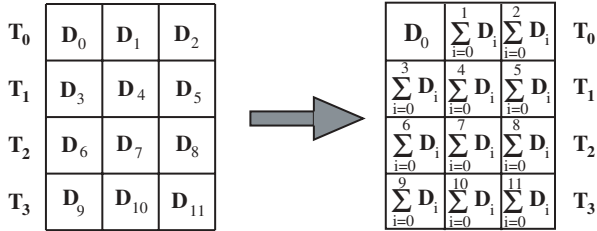
3. If the value of `blk_size` passed to `upc_all_prefix_reduceT` is zero, `upc_all_prefix_reduceT` treats the `src` pointer and the `dst` pointer as if each pointed to a shared memory area of `nelems` elements of type `TYPE` with an indefinite layout qualifier, and therefore had type<sup>7</sup>

```
shared[] TYPE[nelems]
```

4. The phases of the `src` and `dst` pointers are respected when referencing array elements, as specified in items 2 and 3.
5. If the memory areas pointed to by `src` and `dst` overlap, the behavior of this function is undefined.
6. At function exit

```
dst[i] = src[0] ⊕ src[1] ⊕ ⋯ ⊕ src[i]
```

<sup>7</sup>Note that `upc_blocksize(src) == 0` if `src` has this type, so the argument value 0 has a natural connection to the block size of `src`.



**Figure 8** upc\_all\_prefix\_reduce with Addition Operator (the  $D_i$ 's are scalars of type TYPE)

for  $0 \leq i \leq \text{nelems}-1$  and where “ $\oplus$ ” is the operator specified by the variable op.

*Example:* upc\_all\_prefix\_reduce of type long UPC\_ADD (Figure 8):

```
#define BLK_SIZE 3
#define NELEMS 10
shared [BLK_SIZE] long A [NELEMS* THREADS];
shared [BLK_SIZE] long B [NELEMS* THREADS];
// Initialize A.
upc_all_prefix_reduceL(B, A, UPC_ADD, NELEMS* THREADS, BLK_SIZE,
    NULL, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

### 4.3.4 upc\_all\_sort Function

#### Synopsis

```
#include <upc.h>
#include <upc_collective.h>
void upc_all_sort(shared void *A, size_t elem_size, size_t nelems,
    size_t blk_size, int (*func) (shared void *, shared void
*),
    upc_flag_t sync_mode);
elem_size: the size of each element
nelems: the number of elements
blk_size: the number of elements in a block
```

#### Description

1. The function upc\_all\_sort takes a shared array A of nelems elements of size elem\_size bytes each and sorts them in place in ascending order using the function func to compare elements.
2. If the value of blk\_size passed to upc\_all\_sort is greater than zero, upc\_all\_sort treats the array A as if it had blocking factor blk\_size.

3. If the value of `blk_size` passed to `upc_all_sort` is zero, `upc_all_sort` treats the array `A` as if it had an indefinite layout qualifier.
4. The phase of the `A` is respected when referencing array elements, as specified in items 2 and 3.
5. The function `func` shall return an integer less than, equal to, or greater than zero if the first argument is considered to be, respectively, less than, equal to, or greater than the second.
6. The function `upc_all_sort` performs a *stable* sort; that is, elements that compare as equal are not reordered.

*Example:* `upc_all_sort:`

```
#define NELEMS 10
shared [NELEMS] int A [NELEMS* THREADS];

int it_int(shared void *x, shared void *y)
{
    int x_val = * (shared int *)x, y_val = * (shared int *)y;
    return x_val > y_val ? -1 : x_val < y_val ? 1 : 0;
}

// Initialize A.
upc_barrier;
upc_all_sort(A, sizeof(int), NELEMS* THREADS, NELEMS,
             lt_int, UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
upc_barrier;
```

## ACKNOWLEDGMENTS

Thanks go to many in the UPC community for their interest and helpful comments, particularly Dan Bonachea, Bill Carlson, Jason Duell, and Brian Wibecan. Tarek El-Ghazawi's efforts in organizing workshops provided an important forum for discussion of these specifications. Thanks also go to Lauren Smith for her efforts to support the development of this specification.

## REFERENCES

1. E. Wiebel and D. Greenberg, *Collective Operations in UPC*, Technical Report CCS TR-02-159, IDA Center for Computing Sciences, March 2002.
2. T. El-Ghazawi, W. Carlson, and J. Draper, *UPC Language Specifications v1.1.1*, technical report, George Washington University and IDA Center for Computing Sciences, <http://www.gwu.edu/~upc/documentation.html>, October 7, 2003.



# UPC-IO Specifications, v1.0

Tarek El-Ghazawi, François Cantonnet, Proshanta Saha, Rajeev Thakur,  
Rob Ross, and Dan Bonachea

### 3 Terms, Definitions, and Symbols

- 3.1 Collective
- 3.2 Single-Valued
- 3.3 List-Based File Access
- 3.4 File Pointer-Based Access
- 3.5 Synchronous I/O Call
- 3.6 Asynchronous I/O Call
- 3.7 Consistency Semantics
- 3.8 Atomicity Semantics

### 7 Library

- 7.3 UPC Parallel I/O `<upc_io.h>`
  - 7.3.0 Background
    - 7.3.0.1 File Accessing and File Pointers
    - 7.3.0.2 Synchronous and Asynchronous I/O
    - 7.3.0.3 Consistency and Atomicity Semantics
    - 7.3.0.4 File Interoperability
  - 7.3.1 Predefined Types
    - 7.3.1.1 `upc_off_t` Type
    - 7.3.1.2 `upc_file_t` Type
    - 7.3.1.3 `upc_flag_t` Type
    - 7.3.1.4 `upc_local_memvec_t` Type
    - 7.3.1.5 `upc_shared_memvec_t` Type
    - 7.3.1.6 `upc_filevec_t` Type
    - 7.3.1.7 `upc_hint_t` Type

### 7.3.2 UPC File Operations

- 7.3.2.1 upc\_all\_fopen Function
- 7.3.2.2 upc\_all\_fclose Function
- 7.3.2.3 upc\_all\_fsync Function
- 7.3.2.4 upc\_all\_fseek Function
- 7.3.2.5 upc\_all\_fset\_size Function
- 7.3.2.6 upc\_all\_fget\_size Function
- 7.3.2.7 upc\_all\_fpreallocate Function
- 7.3.2.8 upc\_all\_fcntl Function

### 7.3.3 Reading Data

- 7.3.3.1 upc\_all\_fread\_local Function
- 7.3.3.2 upc\_all\_fread\_shared Function

### 7.3.4 Writing Data

- 7.3.4.1 upc\_all\_fwrite\_local Function
- 7.3.4.2 upc\_all\_fwrite\_shared Function

### 7.3.5 List I/O

- 7.3.5.1 upc\_all\_fread\_list\_local Function
- 7.3.5.2 upc\_all\_fread\_list\_shared Function
- 7.3.5.3 upc\_all\_fwrite\_list\_local Function
- 7.3.5.4 upc\_all\_fwrite\_list\_shared Function

### 7.3.6 Asynchronous I/O

- 7.3.6.1 upc\_all\_fread\_local\_async Function
- 7.3.6.2 upc\_all\_fread\_shared\_async Function
- 7.3.6.3 upc\_all\_fwrite\_local\_async Function
- 7.3.6.4 upc\_all\_fwrite\_shared\_async Function
- 7.3.6.5 upc\_all\_fread\_list\_local\_async Function
- 7.3.6.6 upc\_all\_fread\_list\_shared\_async Function
- 7.3.6.7 upc\_all\_fwrite\_list\_local\_async Function
- 7.3.6.8 upc\_all\_fwrite\_list\_shared\_async Function
- 7.3.6.9 upc\_all\_fwait\_async Function
- 7.3.6.10 upc\_all\_ftest\_async Function

## Appendix: Future Library Directions

### References

## 3 TERMS, DEFINITIONS, AND SYMBOLS

Some of the following definitions are repeated from the UPC Language Specifications [1] for self-containment and clarity of the functions defined here.

### 3.1 Collective

A collective is a constraint placed on some language operations that requires evaluation of such operations to be matched across all threads. The behavior of collective operations is undefined unless all threads execute the same sequence of collective operations.<sup>1</sup>

### 3.2 Single-Valued

A single-valued operand to a collective operation has the same value on every thread. The behavior of the operation is otherwise undefined.

### 3.3 List-Based File Access

File accesses are done using explicit offsets and sizes of data. Noncontiguous accesses may be performed using lists of explicit offsets and lengths in the file.

### 3.4 File Pointer-Based Access

File accesses are done using either individual or common file pointers, where individual file pointers provide a means for each thread to control its access to the file independently, and common file pointers provide a means for all the threads to access the file with a single, collective file pointer.

### 3.5 Synchronous I/O Call

Synchronous I/O calls block and wait until the corresponding I/O operation is completed.

### 3.6 Asynchronous I/O Call

Asynchronous I/O calls start an I/O operation and return immediately, and must later be completed using a synchronization function. Only one outstanding asynchronous operation is allowed on a UPC-IO file handle at any given time.

### 3.7 Consistency Semantics

Consistency semantics define when data written to a file by a thread is visible to other threads. The consistency semantics also define the outcome in the case of

<sup>1</sup>A collective operation need not provide any actual synchronization between threads unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any legal program. Some implementations may include unspecified synchronization between threads within collective operations, but programs must not rely on unspecified synchronization for correctness.



overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions.

### 3.8 Atomicity Semantics

Atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file and some of the writes overlap each other.

## 7 LIBRARY

### 7.3 UPC Parallel I/O <upc\_io.h>

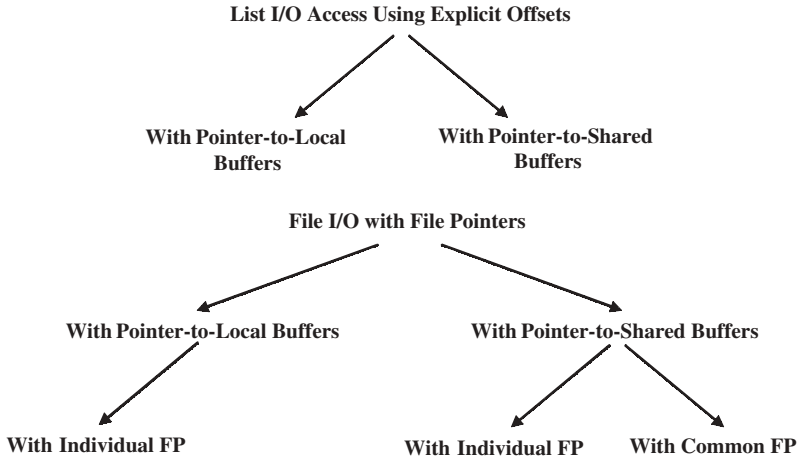
This section provides the UPC parallel extensions of Section 7.19 of [2]. All the characteristics of library functions described in Section 7.1.4 of [2] apply to these as well.

#### *Common Constraints*

1. All UPC-IO functions are collective and must be called by all threads collectively. (See Section 3 of the UPC Specification [1] for a definition of collective.<sup>2</sup>)
2. If a program calls `exit`, `upc_global_exit`, or returns from `main` with a UPC file still open, the file will be closed automatically at program termination, and the effect will be equivalent to `upc_all_fclose` being called on implicitly the file.
3. If a program attempts to read past the end of a file, the read function will read data up to the end of file and return the number of bytes actually read, which may be less than the amount requested.
4. Writing past the end of a file increases the file size.
5. If a program seeks to a location past the end of a file and writes starting from that location, the data in the intermediate (unwritten) portion of the file is undefined. For example, if a program opens a new file (of size 0 bytes), seeks to offset 1024, and writes some data beginning from that offset, the data at offsets 0 through 1023 is undefined. Seeking past the end of file and performing a write immediately causes the current file size to be extended up to the end of the write. However, just seeking past the end of file or attempting to read past the end of file, without a write, does not extend the file size.

<sup>2</sup>Note that *collective* does not necessarily imply barrier synchronization. The synchronization behavior of the UPC-IO data movement library functions is controlled explicitly by using the `sync_mode` flag argument (see Section 7.3.1.3 for details).

6. All “shared void \*” pointers passed to the I/O functions (as function arguments or indirectly through the list I/O arguments) are treated as if they had a phase field of zero (i.e., the input phase is ignored).
7. All UPC-IO read/write functions take an argument `sync_mode` of type `upc_flag_t`. `sync_mode` values are obtained by performing a bitwise OR of a constant of the form `UPC_IN_XSYNC` and a constant of the form `UPC_OUT_YSYNC`, where `X` and `Y` may be `NO`, `MY`, or `ALL`. The `sync_mode` argument is similar to the `sync_mode` argument used in collective operation functions [3]. The `sync_mode` argument and `upc_flag_t` type are discussed further in Section 7.3.1.3.
8. The arguments to all UPC-IO functions are single-valued (must have the same value on all threads) except where explicitly noted otherwise in the function description. (See Section 3 of the UPC Specification [1] for the definition of single-valued.)
9. UPC-IO, by default, supports weak consistency and atomicity semantics. The default (weak) semantics are as follows. The data written to a file by a thread is guaranteed to be visible to another thread only after all threads have collectively closed or synchronized the file.
10. Writes to a file from a given thread are always guaranteed to be visible to subsequent file reads by the *same* thread, even without an intervening call to close or synchronize the file collectively.
11. Byte-level data consistency is supported.
12. If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined with the weak consistency and atomicity semantics.
13. When reading data being written concurrently by another thread, the data that gets read is undefined with the weak consistency and atomicity semantics.
14. File reads into overlapping locations in a shared buffer in memory using individual file pointers or list I/O functions lead to undefined data in the target buffer under the weak consistency and atomicity semantics.
15. A given file must not be opened at the same time by the POSIX I/O and UPC-IO libraries.
16. Except where noted otherwise, all UPC-IO functions return *non*-single-valued errors; that is, the occurrence of an error need only be reported to at least one thread, and the `errno` value reported to each such thread may differ. When an error is reported to *any* thread, the position of *all* file pointers for the relevant file handle becomes undefined.
17. The error values that UPC-IO functions may set in `errno` are implementation-defined; however, the `perror()` and `strerror()` functions are still guaranteed to work properly with `errno` values generated by UPC-IO.
18. UPC-IO functions cannot be called between `upc_notify` and corresponding `upc_wait` statements.



**Figure 1** UPC-IO File Access Methods

### 7.3.0 Background

**7.3.0.1 File Accessing and File Pointers** Collective UPC-IO accesses (Figure 1) can be done in and out of shared and private buffers; thus, local and shared reads and writes are generally supported. In each of these cases, file pointers could be either common or individual. Note that in UPC-IO, common file pointers cannot be used in conjunction with pointer-to-local buffers. File pointer modes are specified by passing a flag to the collective `upc_all_fopen` function and can be changed using `upc_all_fcntl`. When a file is opened with the common file pointer flag, all threads share a common file pointer. When a file is opened with the individual file pointer flag, each thread gets its own file pointer.

UPC-IO also provides file-pointer-independent list file accesses by specifying explicit offsets and sizes of data that is to be accessed. List IO can also be used with either pointer-to-local buffers or pointer-to-shared buffers.

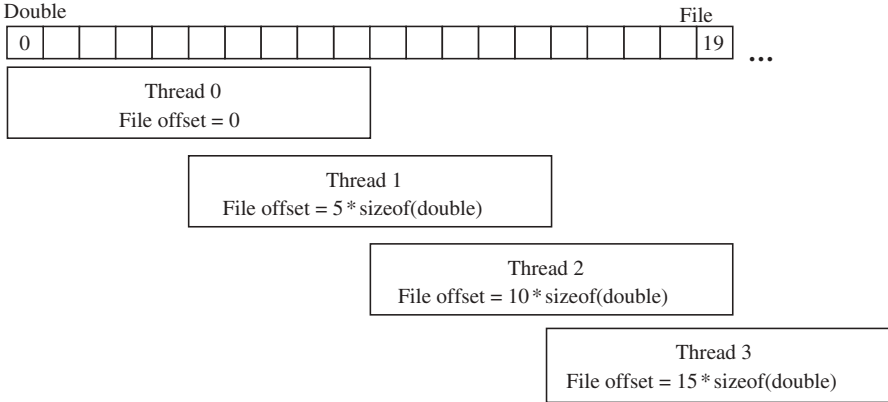
Examples 1 through 3 give typical instances of UPC-IO usage. Error checking is omitted for brevity.

#### **Example 1**

```

double buffer[10]; // and assuming a total of 4 THREADS
upc_file_t *fd;

fd = upc_all_fopen("file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL);
upc_all_fseek(fd, 5*MYTHREAD*sizeof(double), UPC_SEEK_SET);
upc_all_fread_local(fd, buffer, sizeof(double), 10,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
upc_all_fclose(fd);
  
```



**Figure 2** Collective Read into Private Buffers Can Provide a Canonical File View

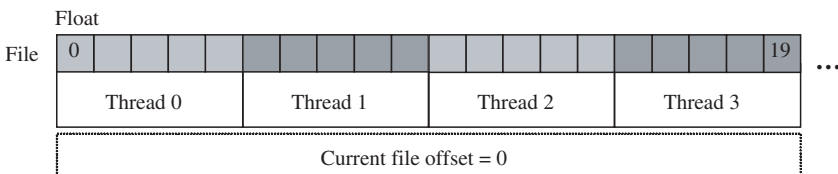
Example 1 and Figure 2 illustrate a collective read operation using individual file pointers. Each thread reads a block of data into a private buffer from a particular thread-specific offset.

Example 2 and Figure 3 illustrate a collective read operation using a common file pointer. The data read is stored into a shared buffer having a block size of five elements. The user selects the type of file pointer at file-open time. The user can select either individual file pointers by passing the flag `UPC_INDIVIDUAL_FP` to the function `upc_all_fopen`, or the common file pointer by passing the flag `UPC_COMMON_FP` to `upc_all_fopen`.

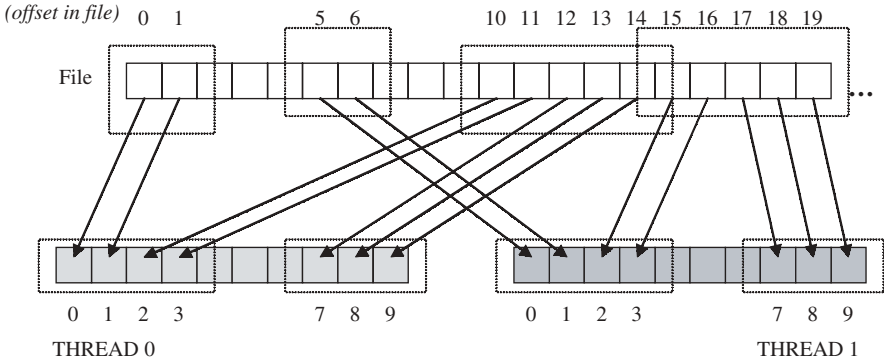
### Example 2

```
shared[5] float buffer[20]; // and assuming a total of 4 static THREADS
upc_file_t *fd;
```

```
fd = upc_all_fopen("file", UPC_RDONLY | UPC_COMMON_FP, 0, NULL);
upc_all_fread_shared(fd, buffer, upc_blocksizeof(buffer),
    upc_elemsizeof(buffer), 20, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
/* or equivalently:
* upc_all_fread_shared(fd, buffer, 5, sizeof(float), 20,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
*/
```



**Figure 3** Collective Read into a Blocked Shared Buffer Can Provide a Partitioned File View



**Figure 4** List I/O Read of Noncontiguous Parts of a File to Private Noncontiguous Buffers

Example 3 and Figure 4 illustrate a collective list I/O read operation. The list I/O functions allow the user to specify noncontiguous accesses both in memory and file in the form of lists of explicit offsets and lengths in the file and explicit address and lengths in memory. None of the file pointers are used or updated in this case.

**Example 3**

```

upc_local_memvec_t memvec [2];
upc_filevec_t filevec [2];
upc_file_t *fd;
char buffer [12];

fd = upc_all_fopen ("file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL);
memvec [0] .baseaddr = &buffer [0];
memvec [0] .len = 4;
memvec [1] .baseaddr = &buffer [7];
memvec [1] .len = 3;
filevec [0] .offset = MYTHREAD* 5;
filevec [0] .len = 2;
filevec [1] .offset = 10+MYTHREAD* 5;
filevec [1] .len = 5;

upc_all_fread_list_local (fd, 2, &memvec, 2, &filevec,
    UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);

```

**7.3.0.2 Synchronous and Asynchronous I/O** I/O operations can be synchronous (blocking) or asynchronous (nonblocking). Whereas synchronous calls are quite simple and easy to use from a programming point of view,

asynchronous operations allow the overlapping of computation and I/O to achieve improved performance. Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an asynchronous call starts an I/O operation and returns immediately. Thus, the executing process can turn its attention to other processing needs while the I/O is progressing.

UPC-IO supports both synchronous and asynchronous I/O functionality. The asynchronous I/O functions have the same syntax and basic semantics as their synchronous counterparts, with the addition of the “async” suffix in their names. The asynchronous I/O functions have the restriction that only one (collective) asynchronous operation can be active at a time on a given file handle. That is, an asynchronous I/O function must be completed by calling `upc_all_fctest_async` or `upc_all_fwait_async` before another asynchronous I/O function can be called on the same file handle. This restriction is similar to the restriction MPI-IO [4] has on split-collective I/O functions: Only one split-collective operation can be outstanding on an MPI-IO file handle at any time.

**7.3.0.3 Consistency and Atomicity Semantics** Let us first define what we mean by the terms *consistency semantics* and *atomicity semantics*. The *consistency semantics* define when the data written to a file by a thread is visible to other threads. The *atomicity semantics* define the outcome of operations in which multiple threads write concurrently to a file or shared buffer and some of the writes overlap each other. For performance reasons, UPC-IO uses weak consistency and atomicity semantics by default. The user can select stronger semantics either by opening the file with the flag `UPC_STRONG_CA` or by calling `upc_all_fcntl` with the command `UPC_SET_STRONG_CA_SEMANTICS`.

The default (weak) semantics are as follows. The data written by a thread is only guaranteed to be visible to another thread after all threads have called `upc_all_fclose` or `upc_all_ftime`. (Note that the data *may* be visible to other threads before the call to `upc_all_fclose` or `upc_all_ftime` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_fclose` or `upc_all_ftime`. Byte-level data consistency is supported. So, for example, if thread 0 writes one byte at offset 0 in the file and thread 1 writes one byte at offset 1 in the file, the data from both threads will get written to the file. If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined. Similarly, if one thread tries to read the data being written concurrently by another thread, the data that gets read is undefined. Concurrent in this context means any two read/write operations to the same file handle with no intervening calls to `upc_all_ftime` or `upc_all_fclose`.

For the functions that read into or write from a shared buffer using a common file pointer, the weak consistency semantics are defined as follows. Each call to `upc_all_{fread,fwrite}_shared[_async]` with a common file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each

operation). In other words, NO file reads are guaranteed to see the results of file writes using the common file pointer until after a close or sync under the default weak consistency semantics.

By passing the `UPC_STRONG_CA` flag to `upc_all_fopen` or by calling `upc_all_fcntl` with the command `UPC_SET_STRONG_CA_SEMANTICS`, the user selects strong consistency and atomicity semantics. In this case, the data written by a thread is visible to other threads as soon as the file write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. Overlapping writes to a file in a single (list I/O) write function on a single thread are not permitted (see Section 7.3.5). While strong consistency and atomicity semantics are selected on a given file handle, the `sync_mode` argument to all `fread/fwrite` functions on that handle is ignored and always treated as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`.

The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions. By default, the data in the overlapping space is undefined. If the user selects strong consistency and atomicity, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order. Overlapping reads into memory buffers in a single (list I/O) read function on a single thread are not permitted (see Section 7.3.5).

Note that in strong consistency and atomicity mode, atomicity is guaranteed at the UPC-IO function level. The entire operation specified by a single function is performed atomically, regardless of whether it represents a single contiguous read/write or multiple noncontiguous reads or writes as in a list I/O function.

**Example** Consider the following example, in which three threads write data to a file concurrently, each with a single list I/O function. The numbers indicate file offsets and brackets indicate the boundaries of a listed vector. Each thread writes its own thread id as the data values:

```
thread 0: {1 2 3} {5 6 7 8}
thread 1: {0 1 2} {3 4 5}
thread 2: {4 5 6} {8 9 10 11}
```

With the default weak semantics, the results in the overlapping locations are undefined. Therefore, the result in the file would be the following, where `x` represents undefined data:

```
File: 1 x x x x x x 0 x 2 2 2
```

That is, the data from thread 1 is written at location 0, the data from thread 0 is written at location 7, and the data from thread 2 is written at locations 9, 10, and 11, because none of these locations had overlapping writes. All other locations had overlapping writes, and consequently, the result at those locations is undefined.

If the file were opened with the `UPC_STRONG_CA` flag, strong consistency and atomicity semantics would be in effect. The result, then, would depend on the order in which the writes from the three threads actually occurred. Since six different orderings are possible, there can be six outcomes. Let us assume, for example, that the ordering was the write from thread 0, followed by the write from thread 2, and then the write from thread 1. The (list I/O) write from each thread happens atomically. Therefore, for this ordering, the result would be

```
File: 1 1 1 1 1 1 2 0 2 2 2 2
```

We note that if instead of using a single list I/O function each thread used a separate function to write each contiguous portion, there would be six write functions, two from each thread, and the atomicity would be at the granularity of the write operation specified by each of those functions.

**7.3.0.4 File Interoperability** UPC-IO does not specify how an implementation may store the data in a file on the storage device. Accordingly, it is implementation-defined whether or not a file created by UPC-IO can be accessed directly by using C/POSIX I/O functions. However, the UPC-IO implementation must specify how the user can retrieve the file from the storage system as a linear sequence of bytes, and vice versa. Similarly, the implementation must specify how familiar operations, such as the equivalent of POSIX `Is`, `cp`, `rm`, and `mv`, can be performed on the file.

## 7.3.1 Predefined Types

### 7.3.1.1 `upc_off_t` Type

#### *Synopsis*

```
#include <upc_io.h>
```

```
upc_off_t myOffset;
```

*Description* `upc_off_t` is a signed integral type that is capable of representing the size of the largest file supported by the implementation.

### 7.3.1.2 `upc_file_t` Type

#### *Synopsis*

```
#include <upc_io.h>
```

```
upc_file_t *myFile;
```



*Description* An opaque shared data type of incomplete type (as defined in Section 6.2.5 of [2]) that represents an open file handle:

```
upc_file_t
```

### *Constraints*

1. `upc_file_t` objects are always manipulated via a pointer (i.e., `upc_file_t *`).
2. `upc_file_t` is a shared data type. It is legal to pass a (`upc_file_t *`) across threads, and two pointers to `upc_file_t` that reference the same logical file handle will always compare equal.

*Advice to implementors* The definition of `upc_file_t` does not restrict the implementation to store all its metadata with affinity to one thread. Each thread can still have local access to its metadata. For example, the following is a simple approach an implementation could use:

```
/* for a POSIX-based implementation */
typedef int my_internal_filehandle_t;

#ifdef _UPC_INTERNAL
    typedef struct _local_upc_file_t {
        my_internal_filehandle_t fd;
        ... other metadata ...
    } local_upc_file_t;
#else
    struct _local_upc_file_t;
#endif

typedef shared struct _local_upc_file_t upc_file_t;

upc_file_t *upc_all_fopen(...) {

    upc_file_t *handles =
        upc_all_alloc(THREADS, sizeof(upc_file_t));

    /* get my handle */
    upc_file_t *myhandle = &(handles [MYTHREAD]);

    /* cast to a pointer-to-local */
    local_upc_file_t* mylocalhandle = (local_upc_file_t*)myhandle;
```

```

/* setup my metadata using pointer-to-local */
mylocalhandle->fd = open(...);

...

return handles;
}

```

The basic idea is that the “handle” exposed to the user actually points to a cyclic, distributed array. As a result, each thread has easy, local access to its own internal handle metadata with no communication, while maintaining the property that the handle that UPC-IO exposes to the client is a single-valued pointer-to-shared. An additional advantage is that a thread can directly access the metadata for other threads, which may occasionally be desirable in the implementation.

### 7.3.1.3 `upc_flag_t` Type

#### *Synopsis*

```

#include <upc_io.h>

upc_flag_t sync_mode;

```

**Description** The `sync_mode` argument is similar to the corresponding argument in collective operation functions [3].

1. If the `sync_mode` has the value (`UPC_IN_XSYNC` | `UPC_OUT_YSYNC`), then if X is

- NO the function may begin to read or write data when the first thread has entered the I/O function call
- MY the function may begin to read or write only data that has affinity to threads that have entered the collective function call
- ALL the function may begin to read or write data only after all threads have entered the collective function call<sup>3</sup>

and if Y is

- NO the function may read and write data until the last thread has returned from the collective function call

<sup>3</sup>`UPC_IN_ALLSYNC` requires the collective I/O function to guarantee that after all threads have entered the collective function call, all threads will read the same values of the input data.

- MY the function call may return in a thread only after all reads and writes of data with affinity to the thread are complete<sup>4</sup>
  - ALL the function call may return only after all reads and writes of data are complete<sup>5</sup>
2. UPC\_IN\_XSYNC alone is equivalent to (UPC\_IN\_XSYNC | UPC\_OUT\_ALLSYNC), where X is NO, MY, or ALL.
  3. UPC\_OUT\_XSYNC alone is equivalent to (UPC\_IN\_ALLSYNC | UPC\_OUT\_XSYNC), where X is NO, MY, or ALL.
  4. 0 is equivalent to (UPC\_IN\_ALLSYNC | UPC\_OUT\_ALLSYNC).
  5. In the `sync_mode` definitions above, “data” refers exclusively to data residing in user-owned memory buffers passed as arguments to the library function. In other words, the `sync_mode` flag only governs the behavior of library accesses to memory locations in user-provided buffers—it does not restrict the behavior of read/write operations on the storage medium or any buffer memory internal to the library implementation.
  6. When applied to the `async` variants of the `fread/fwrite` functions, the semantics of these flags should be interpreted as follows: Constraints that reference entry to a function call correspond to entering the `fread_async-/fwrite_async` call that initiates the asynchronous operation, and constraints that reference returning from a function call correspond to returning from the `upc_all_fwait_async()` or successful `upc_all_ftest_async()` call that completes the asynchronous operation. Also, note that all the `sync_mode` flags that govern an asynchronous operation are passed to the library during the asynchronous initiation call.
  7. The `sync_mode` flag is included even on the `fread/fwrite_local` functions (which take a pointer-to-local as the buffer argument) in order to provide well-defined semantics for the case where one or more of the pointer-to-local arguments references a shared object (with local affinity). In the case where all of the pointer-to-local arguments in a given call reference only private objects, the `sync_mode` flag provides no useful additional guarantees and is recommended to be passed as `UPC_IN_NOSYNC|UPC_OUT_NOSYNC` to maximize performance.

<sup>4</sup>UPC\_OUT\_MYSYNC requires the collective I/O function to guarantee that after a thread returns from the collective function call, the thread will not read any earlier values of the output data with affinity to that thread.

<sup>5</sup>UPC\_OUT\_ALLSYNC requires the collective I/O function to guarantee that after a thread returns from the collective function call, the thread will not read any earlier values of the output data. UPC\_OUT\_ALLSYNC is not required to provide an “implied” barrier. For example, if the entire collective I/O operation has been completed by a certain thread before some other threads have reached their corresponding function calls, that thread may exit its call.

### 7.3.1.4 upc\_local\_memvec\_t Type

#### *Synopsis*

```
#include <upc_io.h>

upc_local_memvec_t myLocalMemoryVector;
```

**Description** upc\_local\_memvec\_t is defined as follows:

```
typedef struct {
    void *baseaddr;
    size_t len;
} upc_local_memvec_t;
```

baseaddr and len specify a contiguous memory region in terms of the base address and length in bytes. len may be zero, in which case that entry is ignored.

### 7.3.1.5 upc\_shared\_memvec\_t Type

#### *Synopsis*

```
#include <upc_io.h>

upc_shared_memvec_t mySharedMemoryVector;
```

**Description** upc\_shared\_memvec\_t is defined as follows:

```
typedef struct {
    shared void *baseaddr;
    size_t blocksize;
    size_t len;
} upc_shared_memvec_t;
```

baseaddr and len specify a shared memory region in terms of the base address and length in bytes. len may be zero, in which case that entry is ignored. blocksize is the block size of the shared buffer in bytes. A blocksize of 0 indicates an indefinite blocking factor.

### 7.3.1.6 upc\_filevec\_t Type

#### *Synopsis*

```
#include <upc_io.h>

upc_filevec_t myFileVector;
```

*Description* `upc_filevec_t` is defined as follows:

```
typedef struct {
    upc_off_t offset;
    size_t len;
} upc_filevec_t;
```

`offset` and `len` specify a contiguous region in the file in terms of the starting offset in the file in bytes and the length in bytes.

### 7.3.1.7 `upc_hint_t` Type

#### *Synopsis*

```
#include <upc_io.h>
upc_hint_t myHint;
```

#### *Description*

1. `upc_hint_t` is defined as follows:

```
typedef struct {
    const char *key;
    const char *value;
} upc_hint_t;
```

2. UPC-IO supports a number of predefined hints. An implementation is free to support additional hints. An implementation is free to ignore any hint provided by the user. Implementations should *silently* ignore any hints they do not support or recognize. The predefined hints and their meanings are defined below. An implementation is not required to interpret these hint keys, but if it does interpret the hint key, it must provide the functionality described. For each hint name introduced, we describe the type of the hint value and its meaning. All hints are single-valued character strings (the content is single-valued, not the location).
  - `access_style` (comma-separated list of strings): indicates the manner in which the file is expected to be accessed. The hint value is a comma-separated list of any of the following: “`read_once`,” “`write_once`,” “`read_mostly`,” “`write_mostly`,” “`sequential`,” and “`random`.” Passing such a hint does not place any constraints on how the file may actually be accessed by the program, although accessing the file in a way that is different from the specified hint may result in lower performance.
  - `collective_buffering` (boolean): specifies whether the application may benefit from collective buffering optimizations. Legal values for this key are “`true`” and “`false`.” Collective buffering parameters can be further directed via additional hints: `cb_buffer_size` and `cb_nodes`.

- `cb_buffer_size` (decimal integer): specifies the total buffer space that the implementation can use on each thread for collective buffering.
- `cb_nodes` (decimal integer): specifies the number of target threads or I/O nodes to be used for collective buffering.
- `file_perm` (string): specifies the file permissions to use for file creation. The set of legal values for this key is implementation-defined.
- `io_node_list` (comma-separated list of strings): specifies the list of I/O devices that should be used to store the file and is relevant only when the file is created.
- `nb_proc` (decimal integer): specifies the number of threads that will typically be used to run programs that access this file and is relevant only when the file is created.
- `striping_factor` (decimal integer): specifies the number of I/O devices that the file should be striped across and is relevant only when the file is created.
- `start_io_device` (decimal integer): specifies the number of the first I/O device from which to start striping the file and is relevant only when the file is created.
- `striping_unit` (decimal integer): specifies the striping unit to be used for the file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

### 7.3.2 UPC File Operations

#### *Common Constraints*

1. When a file is opened with an individual file pointer, each thread will get its own file pointer and advance through the file at its own pace.
2. When a common file pointer is used, all threads positioned in the file will be aligned with that common file pointer.
3. Common file pointers cannot be used in conjunction with pointers-to-local (and hence cannot operate on private objects).
4. No function in this section may be called while an asynchronous operation is pending on the file handle, except where otherwise noted.

#### 7.3.2.1 `upc_all_fopen` Function

##### *Synopsis*

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

```
upc_file_t *upc_all_fopen(const char *fname,
                          int flags,
                          size_t numhints,
                          upc_hint_t const *hints)
```

### Description

1. `upc_all_fopen` opens the file identified by the file name `fname` for I/O operations.
2. The `flags` parameter specifies the access mode. The valid flags and their meanings are listed below. Of these flags, exactly one of `UPC_RDONLY`, `UPC_WRONLY`, or `UPC_RDWR`, and one of `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP`, must be used. Other flags are optional. Multiple flags can be combined by using the bitwise OR operator (`|`), and each flag has a unique bitwise representation that can be tested unambiguously using the bitwise AND operator (`&`).
  - `UPC_RDONLY`: opens the file in read-only mode.
  - `UPC_WRONLY`: opens the file in write-only mode.
  - `UPC_RDWR`: opens the file in read/write mode.
  - `UPC_INDIVIDUAL_FP`: uses an individual file pointer for all file accesses (other than list I/O).
  - `UPC_COMMON_FP`: uses the common file pointer for all file accesses (other than list I/O).
  - `UPC_APPEND`: sets the *initial* position of the file pointer to end of file. (The file pointer is not moved to the end of file after each read/write.)
  - `UPC_CREATE`: creates the file if it does not already exist.
  - `UPC_EXCL`: used in conjunction with `UPC_CREATE`. The open will fail if the file already exists.
  - `UPC_STRONG_CA`: sets strong consistency and atomicity semantics.
  - `UPC_TRUNC`: opens the file and truncate it to zero length.
  - `UPC_DELETE_ON_CLOSE`: deletes the file automatically on close.
3. The `UPC_COMMON_FP` flag specifies that all accesses (except for the list I/O operations) will use the common file pointer. The `UPC_INDIVIDUAL_FP` flag specifies that all accesses will use individual file pointers (except for the list I/O operations). Either `UPC_COMMON_FP` or `UPC_INDIVIDUAL_FP` must be specified or `upc_all_fopen` will return an error.
4. The `UPC_STRONG_CA` flag specifies strong consistency and atomicity semantics. The data written by a thread is visible to other threads as soon as the write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the

individual write function from each thread occurred atomically in some (unspecified) order. In the case of overlapping reads into a shared buffer in memory when using individual file pointers or list I/O functions, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order.

5. If the flag `UPC_STRONG_CA` is not specified, weak semantics are provided. The data written by a thread is guaranteed to be visible to another thread only after all threads have called `upc_all_fclose` or `upc_all_fsync`. (Note that the data *may* be visible to other threads before the call to `upc_all_fclose` or `upc_all_fsync` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_fclose` or `upc_all_fsync`. Byte-level data consistency is supported. For the purposes of atomicity and consistency semantics, each call to `upc_all_{fread,fwrite}_shared[_async]` with a common file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each operation).<sup>6</sup>
6. Hints can be passed to the UPC-IO library as an array of key-value pairs<sup>7</sup> of strings. `numhints` specifies the number of hints in the `hints` array; if `numhints` is zero, the `hints` pointer is ignored. The user can free the `hints` array and associated character strings as soon as the open call returns. Each element of the `hints` array is of type `upc_hint_t`.
7. A file on the storage device is in the *open* state from the beginning of a successful open call to the end of the matching successful close call on the file handle. It is erroneous to have the same file *open* simultaneously with two `upc_all_fopen` calls, or with a `upc_all_fopen` call and a POSIX/C `open` or `fopen` call.
8. The user is responsible for ensuring that the file referenced by the `fname` argument refers to a single UPC-IO file. The actual argument passed on each thread may be different because the file name spaces may be different on different threads, but they must all refer to the same logical UPC-IO file.
9. On success, the function returns a pointer to a file handle that can be used to perform other operations on the file.
10. `upc_all_fopen` provides single-valued errors—if an error occurs, the function returns `NULL` on *all* threads and sets `errno` appropriately to the same value on all threads.

<sup>6</sup> In other words, NO reads are guaranteed to see the results of writes using the common file pointer until after a close or sync when `UPC_STRONG_CA` is not specified.

<sup>7</sup> The contents of the key/value pairs passed by all the threads must be single-valued.



### 7.3.2.2 `upc_all_fclose` Function

#### *Synopsis*

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

int upc_all_fclose(upc_file_t *fd);
```

#### *Description*

1. `upc_all_fclose` executes an implicit `upc_all_fsync` on `fd` and then closes the file associated with `fd`.
2. The function returns 0 on success. If `fd` is not valid or if an outstanding asynchronous operation on `fd` has not been completed, the function will return an error.
3. `upc_all_fclose` provides single-valued errors—if an error occurs, the function returns `-1` on ALL threads and sets `errno` appropriately to the same value on all threads.
4. After a file has been closed with `upc_all_fclose`, the file can legally be opened and the data in it can be accessed by using regular C/POSIX I/O calls.

### 7.3.2.3 `upc_all_fsync` Function

#### *Synopsis*

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

int upc_all_fsync(upc_file_t *fd)
```

#### *Description*

1. `upc_all_fsync` ensures that any data that has been written to the file associated with `fd` but not yet transferred to the storage device is transferred to the storage device. It also ensures that subsequent file reads from any thread will see any previously written values (that have not yet been overwritten).
2. There is an implied barrier immediately before `upc_all_fsync` returns.
3. The function returns 0 on success. On error, it returns `-1` and sets `errno` appropriately.

### 7.3.2.4 `upc_all_fseek` Function

#### *Synopsis*

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

```
upc_off_t upc_all_fseek(upc_file_t *fd,
    upc_off_t offset,
    int origin)
```

#### *Description*

1. `upc_all_fseek` sets the current position of the file pointer associated with `fd`.
2. This offset can be relative to the current position of the file pointer, to the beginning of the file, or to the end of the file. The offset can be negative, which allows seeking backward.
3. The origin parameter can be specified as `UPC_SEEK_SET`, `UPC_SEEK_CUR`, or `UPC_SEEK_END`, respectively, to indicate that the offset must be computed from the beginning of the file, the current location of the file pointer, or the end of the file.
4. In the case of a common file pointer, all threads must specify the same offset and origin. In the case of an individual file pointer, each thread may specify a different offset and origin.
5. It is legal to seek past the end of file. It is erroneous to seek to a negative position in the file. See common constraint 5 at the beginning of Section 7.3 for more details.
6. The current position of the file pointer can be determined by calling `upc_all_fseek(fd, 0, UPC_SEEK_CUR)`.
7. On success, the function returns the current location of the file pointer in bytes. If there is an error, it returns `-1` and sets `errno` appropriately.

### 7.3.2.5 `upc_all_fset_size` Function

#### *Synopsis*

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

```
int upc_all_fset_size(upc_file_t *fd,
    upc_off_t size)
```

*Description*

1. `upc_all_fset_size` executes an implicit `upc_all_fsync` on `fd` and resizes the file associated with `fd`.
2. `size` is measured in bytes from the beginning of the file.
3. If `size` is less than the current file size, the file is truncated at the position defined by `size`. The implementation is free to deallocate file blocks located beyond this position.
4. If `size` is greater than the current file size, the file size increases to `size`. Regions of the file written previously are unaffected. The values of data in the new regions in the file (between the old size and `size`) are undefined.
5. If this function truncates a file, it is possible that the individual and common file pointers may point beyond the end of file. This is legal and is equivalent to seeking past the end of file (the Common Rules in Section 5 describe the semantics of seeking past the end of file).
6. It is not specified whether and under what conditions this function actually allocates file space on the storage device. Use `upc_all_fpreallocate` to force file space to be reserved on the storage device.
7. Calling this function does not affect the individual or common file pointers.
8. The function returns 0 on success. On error, it returns `-1` and sets `errno` appropriately.

**7.3.2.6** `upc_all_fget_size` Function*Synopsis*

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

upc_off_t upc_all_fget_size(upc_file_t *fd)
```

*Description* `upc_all_fget_size` returns the current size in bytes of the file associated with `fd` on success. On error, it returns `-1` and sets `errno` appropriately.

**7.3.2.7** `upc_all_fpreallocate` Function*Synopsis*

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

int upc_all_fpreallocate(upc_file_t *fd,
                        upc_off_t size)
```

### Description

1. `upc_all_fpreallocate` ensures that storage space is allocated for the first `size` bytes of the file associated with `fd`.
2. Regions of the file written previously are unaffected. For newly allocated regions of the file, `upc_all_fpreallocate` has the same effect as writing undefined data.
3. If `size` is greater than the current file size, the file size increases to `size`. If `size` is less than or equal to the current file size, the file size is unchanged.
4. Calling this function does not affect the individual or common file pointers.
5. The function returns 0 on success. On error, it returns `-1` and sets `errno` appropriately.

### 7.3.2.8 `upc_all_fcntl` Function

#### Synopsis

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

int upc_all_fcntl(upc_file_t *fd,
                 int cmd,
                 void *arg)
```

#### Description

1. `upc_all_fcntl` performs one of various miscellaneous operations related to the file specified by `fd`, as determined by `cmd`. The valid commands `cmd` and their associated argument `arg` are explained below.
  - `UPC_GET_CA_SEMANTICS`: gets the current consistency and atomicity semantics for `fd`. The argument `arg` is ignored. The return value is `UPC_STRONG_CA` for strong consistency and atomicity semantics and 0 for the default weak consistency and atomicity semantics.
  - `UPC_SET_WEAK_CA_SEMANTICS`: executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the weak consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns `-1` and sets `errno` appropriately.
  - `UPC_SET_STRONG_CA_SEMANTICS`: executes an implicit `upc_all_fsync` on `fd` and sets `fd` to use the strong consistency and atomicity semantics (or leaves the mode unchanged if that mode is already selected). The argument `arg` is ignored. The return value is 0 on success. On error, this function returns `-1` and sets `errno` appropriately.

- `UPC_GET_FP`: gets the type of the current file pointer for `fd`. The argument `arg` is ignored. The return value is either `UPC_COMMON_FP` in case of a common file pointer, or `UPC_INDIVIDUAL_FP` for individual file pointers.
- `UPC_SET_COMMON_FP`: executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `fd` to a common file pointer (or leaves it unchanged if that mode is already selected), and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns `-1` and sets `errno` appropriately.
- `UPC_SET_INDIVIDUAL_FP`: executes an implicit `upc_all_fsync` on `fd`, sets the current file access pointer mechanism for `fd` to an individual file pointer (or leaves the mode unchanged if that mode is already selected), and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns `-1` and sets `errno` appropriately.
- `UPC_GET_FL`: gets all the flags specified during the `upc_all_fopen` call for `fd`, as modified by any subsequent mode changes using the `upc_all_fcntl(UPC_SET_*)` commands. The argument `arg` is ignored. The return value has same format as the `flags` parameter in `upc_all_fopen`.
- `UPC_GET_FN`: gets the file name provided by each thread in the `upc_all_fopen` call that created `fd`. The argument `arg` is a valid (`const char**`) pointing to a (`const char*`) location in which a pointer to file name will be written. Writes a (`const char*`) into `*arg` pointing to the file name in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fcntl` call on that file handle.
- `UPC_GET_HINTS`: retrieves the hints applicable to `fd`. The argument `arg` is a valid (`const upc_hint_t**`) pointing to a (`const upc_hint_t*`) location in which a pointer to the hints array will be written. Writes a (`const upc_hint_t*`) into `*arg` pointing to an array of `upc_hint_t`'s in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fcntl` call on that file handle. The number of hints in the array is returned by the call. The hints in the array may be a subset of those specified at file open time if the implementation ignored some unrecognized or unsupported hints.
- `UPC_SET_HINT`: executes an implicit `upc_all_fsync` on `fd` and sets a new hint to `fd`. The argument `arg` points to one single-valued `upc_hint_t` hint to be applied. If the given hint key has already been applied to `fd`, the current value for that hint is replaced with the value provided. The return value is 0 on success. On error, this function returns `-1` and sets `errno` appropriately.

- `UPC_ASYNC_OUTSTANDING`: returns 1 if there is an asynchronous operation outstanding on `fd`, or 0 otherwise.
2. In case of a nonvalid `fd`, `upc_all_fcntl` returns `-1` and sets `errno` appropriately.
  3. It is legal to call `upc_all_fcntl(UPC_ASYNC_OUTSTANDING)` when an asynchronous operation is outstanding (but it is still illegal to call `upc_all_fcntl` with any other argument when an asynchronous operation is outstanding)

### 7.3.3 Reading Data

*Common Constraint* No function in this section may be called while an asynchronous operation is pending on the file handle.

#### 7.3.3.1 `upc_all_fread_local` Function

##### *Synopsis*

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

ssize_t upc_all_fread_local(upc_file_t *fd,
                           void *buffer,
                           size_t size,
                           size_t nmemb,
                           upc_flag_t sync_mode)
```

##### *Description*

1. `upc_all_fread_local` reads data from a file into a local buffer on each thread.
2. This function can be called only if the current file pointer type is an individual file pointer and the file handle is open for reading.
3. `buffer` is a pointer to an array into which data will be read, and each thread may pass a different value for `buffer`.
4. Each thread reads (`size*nmemb`) bytes of data from the file at the position indicated by its individual file pointer into the buffer. Each thread may pass a different value for `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.
5. On success, the function returns the number of bytes read into the local buffer of the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns `-1` and sets `errno` appropriately.

### 7.3.3.2 upc\_all\_fread\_shared Function

#### Synopsis

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

ssize_t upc_all_fread_shared(upc_file_t *fd,
                             shared void *buffer,
                             size_t blocksize,
                             size_t size,
                             size_t nmemb,
                             upc_flag_t sync_mode)
```

#### Description

1. `upc_all_fread_shared` reads data from a file into a shared buffer in memory.
2. The function can be called when the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for reading.
3. `buffer` is a pointer to an array into which data will be read. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of zero indicates an indefinite blocking factor.
4. In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. Each thread reads  $(size * nmemb)$  bytes of data from the file at the position indicated by its individual file pointer into its buffer. Each thread may pass a different value for `blocksize`, `size`, and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes read by the calling thread, and the individual file pointer of the thread is incremented by that amount.
5. In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter and the same value for `blocksize`, `size`, and `nmemb`. The effect is that  $(size * nmemb)$  bytes of data are read from the file at the position indicated by the common file pointer into the buffer. If `size` or `nmemb` is zero, the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes read by all threads, and the common file pointer is incremented by that amount.
6. If reading with individual file pointers results in overlapping reads into the shared buffer, the result is determined by whether or not the file was opened with the `UPC_STRONG_CA` flag (see Section 7.3.2.1).
7. The function returns `-1` on error and sets `errno` appropriately.

### 7.3.4 Writing Data

*Common Constraint* No function in this section may be called while an asynchronous operation is pending on the file handle.

#### 7.3.4.1 `upc_all_fwrite_local` Function

##### *Synopsis*

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

ssize_t upc_all_fwrite_local (upc_file_t *fd,
                             void *buffer,
                             size_t size,
                             size_t nmemb,
                             upc_flag_t sync_mode)
```

##### *Description*

1. `upc_all_fwrite_local` writes data from a local buffer on each thread into a file.
2. This function can be called only if the current file pointer type is an individual file pointer and the file handle is open for writing.
3. `buffer` is a pointer to an array from which data will be written, and each thread may pass a different value for `buffer`.
4. Each thread writes (`size*nmemb`) bytes of data from the buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for `size` and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O.
5. If any of the writes result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).
6. On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount. On error, it returns `-1` and sets `errno` appropriately.

#### 7.3.4.2 `upc_all_fwrite_shared` Function

##### *Synopsis*

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



```

ssize_t upc_all_fwrite_shared(upc_file_t *fd,
    shared void *buffer,
    size_t blocksize,
    size_t size,
    size_t nmemb,
    upc_flag_t sync_mode)

```

### *Description*

1. `upc_all_fwrite_shared` writes data from a shared buffer in memory to a file.
2. The function can be called if the current file pointer type is either a common file pointer or an individual file pointer. The file handle must be open for writing.
3. `buffer` is a pointer to an array from which data will be written. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in elements (of `size` bytes each). A `blocksize` of zero indicates an indefinite blocking factor.
4. In the case of individual file pointers, the following rules apply: Each thread may pass a different address for the buffer parameter. Each thread writes  $(size * nmemb)$  bytes of data from its buffer to the file at the position indicated by its individual file pointer. Each thread may pass a different value for `blocksize`, `size`, and `nmemb`. If `size` or `nmemb` is zero, the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes written by the calling thread, and the individual file pointer of the thread is incremented by that amount.
5. In the case of a common file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter and the same value for `blocksize`, `size`, and `nmemb`. The effect is that  $(size * nmemb)$  bytes of data are written from the buffer to the file at the position indicated by the common file pointer. If `size` or `nmemb` is zero, the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes written by all threads, and the common file pointer is incremented by that amount.
6. If writing with individual file pointers results in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).
7. The function returns `-1` on error and sets `errno` appropriately.

### **7.3.5 List I/O**

#### *Common Constraints*

1. List I/O functions take a list of addresses/offsets and corresponding lengths in memory and file to read from or write to.

2. List I/O functions can be called regardless of whether the current file pointer type is individual or common.
3. File pointers are not updated as a result of a list I/O read/write operation.
4. The `memvec` argument passed to any list I/O *read* function by a single thread must not specify overlapping regions in memory.
5. The base addresses passed to `memvec` can be in any order.
6. The `filevec` argument passed to any list I/O *write* function by a single thread must not specify overlapping regions in the file.
7. The offsets passed in `filevec` must be in monotonically nondecreasing order.
8. No function in this section may be called while an asynchronous operation is pending on the file handle.
9. No function in this section implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier after calling `upc_all_fread_list_shared` to ensure that the entire shared buffer has been filled up, and similarly, use a barrier before calling `upc_all_fwrite_list_shared` to ensure that the entire shared buffer is up-to-date before being written to the file.
10. For all the list I/O functions, each thread passes an independent set of memory and file vectors. Passing the same vectors on two or more threads specifies redundant work. The file pointer is a single-valued argument, all other arguments to the list I/O functions are NOT single-valued.

### 7.3.5.1 `upc_all_fread_list_local` Function

#### *Synopsis*

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

ssize_t upc_all_fread_list_local(upc_file_t *fd,
                                size_t memvec_entries,
                                upc_local_memvec_t const *memvec,
                                size_t filevec_entries,
                                upc_filevec_t const *filevec,
                                upc_flag_t sync_mode)
```

#### *Description*

1. `upc_all_fread_list_local` reads data from a file into local buffers in memory. The file handle must be open for reading.
2. `memvec_entries` indicates the number of entries in the array `memvec`, and `filevec_entries` indicates the number of entries in the array

filevec. The values may be zero, in which case the memvec or filevec argument is ignored and no locations are specified for I/O.

3. The result is as if data were read in order from the list of locations specified by filevec and placed in memory in the order specified by the list of locations in memvec. The total amount of data specified by memvec must equal the total amount of data specified by filevec.
4. On success, the function returns the number of bytes read by the calling thread. On error, it returns `-1` and sets `errno` appropriately.

### 7.3.5.2 `upc_all_fread_list_shared` Function

#### *Synopsis*

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

ssize_t upc_all_fread_list_shared(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

#### *Description*

1. `upc_all_fread_list_shared` reads data from a file into various locations of a shared buffer in memory. The file handle must be open for reading.
2. `memvec_entries` indicates the number of entries in the array `memvec`, and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be zero, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.
3. The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.
4. If any of the reads from different threads result in overlapping regions in memory, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).
5. On success, the function returns the number of bytes read by the calling thread. On error, it returns `-1` and sets `errno` appropriately.

*Note: With the definition above there is no way to do with explicit offsets the equivalent of `upc_all_fread_shared` using a common file pointer; namely, where*

*all threads specify the same access (same parameters), the data gets read collectively into the shared buffer and the function returns the total amount of data read by all threads.*

### 7.3.5.3 `upc_all_fwrite_list_local` Function

#### *Synopsis*

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

ssize_t upc_all_fwrite_list_local(upc_file_t *fd,
    size_t memvec_entries,
    upc_local_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

#### *Description*

1. `upc_all_fwrite_list_local` writes data from local buffers in memory to a file. The file handle must be open for writing.
2. `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be zero, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.
3. The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.
4. If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).
5. On success, the function returns the number of bytes written by the calling thread. On error, it returns `-1` and sets `errno` appropriately.

### 7.3.5.4 `upc_all_fwrite_list_shared` Function

#### *Synopsis*

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

```

ssize_t upc_all_fwrite_list_shared(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)

```

### *Description*

1. `upc_all_fwrite_list_shared` writes data from various locations of a shared buffer in memory to a file. The file handle must be open for writing.
2. `memvec_entries` indicates the number of entries in the array `memvec`, and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be zero, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.
3. The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the total amount of data specified by `filevec`.
4. If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the current consistency and atomicity semantics mode in effect for `fd` (see Section 7.3.2.1).
5. On success, the function returns the number of bytes written by the calling thread. On error, it returns `-1` and sets `errno` appropriately.

*Note: With the definition above, there is no way to do with explicit offsets the equivalent of `upc_all_fwrite_shared` using a common file pointer; namely, where all threads specify the same access (same parameters), the data gets written collectively from a shared buffer and the function returns the total amount of data written by all threads.*

## **7.3.6 Asynchronous I/O**

### *Common Constraints*

1. Only one asynchronous I/O operation can be outstanding on a UPC-IO file handle at any time. If an application attempts to initiate a second asynchronous I/O operation while one is still outstanding on the same file handle, the behavior is undefined—however, high-quality implementations will issue a fatal error.
2. For asynchronous read operations, the contents of the destination memory are undefined until after a successful `upc_all_fwait_async` or `upc_all_ftest_async` on the file handle. For asynchronous write operations,

the source memory may not safely be modified until after a successful `upc_all_fwait_async` or `upc_all_ftest_async` on the file handle.

3. An implementation is free to block for completion of an operation in the asynchronous initiation call or in the `upc_all_ftest_async` call (or both). High-quality implementations are recommended to minimize the amount of time spent within the asynchronous initiation or `upc_all_ftest_async` call.
4. In the case of list I/O functions, the user may modify or free the lists after the asynchronous I/O operation has been initiated.

### 7.3.6.1 `upc_all_fread_local_async` Function

#### *Synopsis*

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

void upc_all_fread_local_async (upc_file_t *fd,
                               void *buffer,
                               size_t size,
                               size_t nmemb,
                               upc_flag_t sync_mode)
```

#### *Description*

1. `upc_all_fread_local_async` initiates an asynchronous read from a file into a local buffer on each thread.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fread_local`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

### 7.3.6.2 `upc_all_fread_shared_async` Function

#### *Synopsis*

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

void upc_all_fread_shared_async (upc_file_t *fd,
                                shared void *buffer,
                                size_t blocksize,
                                size_t size,
```

```

    size_t nmemb,
    upc_flag_t sync_mode)

```

### *Description*

1. `upc_all_fread_shared_async` initiates an asynchronous read from a file into a shared buffer.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fread_shared`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

#### 7.3.6.3 `upc_all_fwrite_local_async` Function

##### *Synopsis*

```

#include <upc.h>
#include <upc_io.h>

void upc_all_fwrite_local_async(upc_file_t *fd,
    void *buffer,
    size_t size,
    size_t nmemb,
    upc_flag_t sync_mode)

```

### *Description*

1. `upc_all_fwrite_local_async` initiates an asynchronous write from a local buffer on each thread to a file.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fwrite_local`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

#### 7.3.6.4 `upc_all_fwrite_shared_async` Function

##### *Synopsis*

```

#include <upc.h>
#include <upc_io.h>

```

```
void upc_all_fwrite_shared_async(upc_file_t *fd,
                                shared void *buffer,
                                size_t blocksize,
                                size_t size,
                                size_t nmemb,
                                upc_flag_t sync_mode)
```

### *Description*

1. `upc_all_fwrite_shared_async` initiates an asynchronous write from a shared buffer to a file.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fwrite_shared`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

### 7.3.6.5 `upc_all_fread_list_local_async` Function

#### *Synopsis*

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

void upc_all_fread_list_local_async(upc_file_t *fd,
                                    size_t memvec_entries,
                                    upc_local_memvec_t const *memvec,
                                    size_t filevec_entries,
                                    upc_filevec_t const *filevec,
                                    upc_flag_t sync_mode)
```

### *Description*

1. `upc_all_fread_list_local_async` initiates an asynchronous read of data from a file into local buffers in memory.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fread_list_local`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

### 7.3.6.6 `upc_all_fread_list_shared_async` Function

#### *Synopsis*

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



```
void upc_all_fread_list_shared_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

### *Description*

1. `upc_all_fread_list_shared_async` initiates an asynchronous read of data from a file into various locations of a shared buffer in memory.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fread_list_shared`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

### 7.3.6.7 `upc_all_fwrite_list_local_async` Function

#### *Synopsis*

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

```
void upc_all_fwrite_list_local_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_local_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

### *Description*

1. `upc_all_fwrite_list_local_async` initiates an asynchronous write of data from local buffers in memory to a file.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fwrite_list_local`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

### 7.3.6.8 upc\_all\_fwrite\_list\_shared\_async Function

#### *Synopsis*

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

void upc_all_fwrite_list_shared_async(upc_file_t *fd,
    size_t memvec_entries,
    upc_shared_memvec_t const *memvec,
    size_t filevec_entries,
    upc_filevec_t const *filevec,
    upc_flag_t sync_mode)
```

#### *Description*

1. `upc_all_fwrite_list_shared_async` initiates an asynchronous write of data from various locations of a shared buffer in memory to a file.
2. The meaning of the parameters and restrictions is the same as for the blocking function, `upc_all_fwrite_list_shared`.
3. The status of the asynchronous I/O operation that has been initiated can be retrieved by calling `upc_all_ftest_async` or `upc_all_fwait_async`.

### 7.3.6.9 upc\_all\_fwait\_async Function

#### *Synopsis*

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

ssize_t upc_all_fwait_async(upc_file_t *fd)
```

#### *Description*

1. `upc_all_fwait_async` completes the previously issued asynchronous I/O operation on the file handle `fd`, blocking if necessary.
2. It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.
3. On success, the function returns the number of bytes read or written by the asynchronous I/O operation as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns `-1` and sets `errno` appropriately, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

### 7.3.6.10 `upc_all_ftest_async` Function

#### *Synopsis*

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

ssize_t upc_all_ftest_async(upc_file_t *fd,
                           int *flag)
```

#### *Description*

1. `upc_all_ftest_async` tests whether the outstanding asynchronous I/O operation associated with `fd` has completed.
2. If the operation has completed, the function sets `flag=1` and the asynchronous operation becomes no longer outstanding<sup>8</sup>; otherwise, it sets `flag=0`. The same value of `flag` is set on all threads.
3. If the operation was completed, the function returns the number of bytes that were read or written as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns `-1` and sets `errno` appropriately and sets the `flag=1`, and the outstanding asynchronous operation (if any) becomes no longer outstanding.
4. It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.

## APPENDIX: FUTURE LIBRARY DIRECTIONS

We describe here features that will be discussed in future releases of the UPC-IO Specifications (but have explicitly been tabled for the current release).

1. Add support for allowing multiple outstanding asynchronous operations on the same file handle, to allow more aggressive computational overlap with file operations. This can be done by introducing a handle data type such as `upc_all_handle_t` to represent the explicitly nonblocking collective I/O operation in flight, and have this value returned by each async init function and consumed by explicit-handle synchronization functions (e.g., `upc_all_handle_test()` / `upc_all_handle_wait()`). All of the asynchronous initiation functions currently return void, so we could add this new handle return type without breaking existing code. To be backward-compatible with the current interface, the default behavior should remain at the current behavior [allow only a single outstanding async operation, synchronized

<sup>8</sup>This implies that it is illegal to call `upc_all_fwait_async` or `upc_all_ftest_async` immediately after a successful `upc_all_ftest_async` on that file handle.

using `upc_all_ftest_async()/upc_all_fwait_async()`] and we can use a new `upc_all_fcntl` setting to enable multioperation handle-based async I/O as described here. We may even want to use the same handle data type and sync functions for explicitly nonblocking UPC collectives.

An alternative approach to allowing multiple outstanding asynchronous operations on the same file handle would be an implicit-handle approach, where we keep the current interface and simply lift the restriction that only one asynchronous operation can be in flight per handle. This approach offers the client less flexibility in synchronization (because the only choice is to `sync` all outstanding operations rather than a particular subset), but it may be an acceptable compromise. However, we would have to think about how errors would be reported by synchronization functions that complete more than one operation.

Regardless of the design chosen, an important semantic issue that must be resolved when more than one async call can be in flight is specifying exactly when and how the file pointer is updated by an async operation, especially in the presence of errors or reading to the EOF. The semantics of the second and subsequent async I/O operations are not well defined unless we specify how the file pointer is affected by the async I/O operations already in flight. One possibility for sidestepping this issue is to only allow multiple outstanding async I/O operations of the list I/O variety, which are completely independent of the problematic file pointer.

2. `upc_all_fcntl` currently provides the means to change most of the interesting `upc_all_fopen` flags in effect for the given file handle. The only `upc_all_fopen` flag that persists as an attribute of the file handle and currently cannot be changed after open is the read-only/write-only/read-write status of the file handle. Do we want to support changing this via a `upc_all_fcntl`? Is it a useful capability? What are the implementation issues? (At worst it would seem that this could always be implemented with a close and reopen.) Note that C99 provides this capability via `freopen()`, and UPC-IO currently has no equivalent—however, the C99 semantics are too weak to be portably reliable: “If filename is a null pointer, the `freopen()` function attempts to change the mode of the stream to that specified by mode, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.” It is also unclear from the spec what the required behavior of the file pointer is on such an `freopen()`. If we decide to provide this capability, we should be less wishy-washy about the semantics to ensure that it is portably usable.

## REFERENCES

1. Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper, *UPC Language Specifications v1.1.1*, <http://upc.gwu.edu>, October 2003.

2. *Programming Languages—C*, ISO/IEC 9899:1999(E), ISO/IEC, May 2000.
3. Elizabeth Wiebel, David Greenberg, and Steven Seidel, *UPC Collective Operations Specification v1.0*, [http://www.gwu.edu/upc/docs/UPC\\_Coll\\_Spec\\_v1.0.pdf](http://www.gwu.edu/upc/docs/UPC_Coll_Spec_v1.0.pdf), December 2003.
4. *MPI-2: Extensions to the Message-Passing Interface*, Message Passing Interface Forum, July 18, 1997.

## How to Compile and Run UPC Programs

In this appendix we give a few examples of how to compile and run UPC programs on a sample of the compilers available. These examples are intended to give the reader an idea of how to do so in general. Readers are encouraged to examine the programming manual of their target machines to become familiar with how to optimize the compilation and execution of their code through compiler switches and shell variables. Many of these manuals also provide extensive coverage of the performance tuning and debugging tools available on their target platform.

### COMPILING AND RUNNING ON THE CRAY X1 UPC COMPILER

To compile and run with a fixed number of threads (here, four THREADS):

```
cc -h upc -X 4 -o helloworld1 helloworld1.upc  
aprun -n 4. /helloworld1
```

To compile and run without specifying the number of threads at compile time:

```
cc -h upc -o helloworld1 helloworld1.upc  
aprun -n 4. /helloworld1
```

**COMPILING AND RUNNING ON THE HP-UPC COMPILER**

To compile and run with a fixed number of threads (here, four THREADS):

```
upc -O2 -fthreads 4 -o helloworld1 helloworld1.upc
prun -n 4. /helloworld1
```

To compile and run without specifying the number of threads at compile time:

```
upc -O2 -o helloworld1 helloworld1.upc
prun -n 4. /helloworld1
```

**COMPILING AND RUNNING ON THE SGI-INTREPID GCC-UPC COMPILER**

To compile and run with a fixed number of threads (here, four THREADS):

```
upc -x upc -fupc-threads-4 -O2 -o helloworld1 helloworld1.upc
./vect_add
```

To compile and run without specifying the number of threads at compile time:

```
upc -x upc -O2 -o helloworld1 helloworld1.upc
./vect_add -fupc-threads-4
```

**COMPILING AND RUNNING ON THE BERKELEY UPC COMPILER**

To compile and run with a fixed number of threads (here, four THREADS):

```
upcc -O -T 4 -o helloworld1 helloworld1.upc
upcrun. /helloworld1
```

To compile and run without specifying the number of threads at compile time:

```
upcc -O -o helloworld1 helloworld1.upc
upcrun -n 4. /helloworld1
```

## Quick UPC Reference

### 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 DECLARATIONS

#### Shared Objects

Shared variables are declared using the type qualifier “shared.” Shared objects must be declared statically (i.e., either as global variables or with the keyword *static*).

#### *Examples 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** are allocated in thread 0:

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

---

*UPC: Distributed Shared Memory Programming*, by Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick  
Published 2005 John Wiley & Sons, Inc.



Distribute the elements in round-robin fashion by chunks of two elements: `a[0]` and `a[1]` are allocated in thread 0; `a[2]` and `a[3]` in thread 1; ...:

```
shared [2] int a[100];
```

### Shared Pointers

Pointer to shared object:

```
shared int* p;
```

Shared pointer to shared object:

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

Equivalent of `memset`:

```
upc_memset(dst, char, size)
```

Assign a block of characters to shared memory.

### LOCKS

//Dynamic lock collectively allocated:

```
{
  upc_lock_t *l;
  l = upc_all_lock_alloc();

  //...
  upc_lock (l);
  // protected section
  upc_unlock (l);

  if ( upc_lock_attempt (l)
    {
      //do something if l currently unlocked
    }

  //deallocates the lock
  upc_lock_free (l);
}
```

//Dynamic lock globally allocated:

```
upc_lock_t *l;
```

```
{
  if (MYTHREAD == 3)
    l = upc_global_lock_alloc();
}
```

## GENERAL UTILITIES

Terminate the UPC program with exit status **status**:

```
upc_global_exit(status);
```

## WORK SHARING

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

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

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

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

Distributes the iterations by consecutive chunks:

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

## SYNCHRONIZATION

### Memory Consistency

These include files set to which consistency model, strict or relaxed, is used for the entire program.

```
#include "upc_strict.h"
```

or

```
"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** are made using the relaxed consistency model:

```
shared relaxed int i;
```

All accesses to **i** are made using the relaxed consistency model:

```
relaxed shared int i;
```

All accesses to **i** are made using the strict consistency model:

```
strict shared int i;
```

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

```
upc_fence;
```

## Barriers

Synchronizes the program globally:

```
upc_barrier [value];
```

**value** is an integer.

```
//Before the barrier:
```

```
upc_notify [value];
```

```
// Nonsynchronized statements relative to this ongoing barrier:
```

```
upc_wait [value];
```

```
// After the barrier:
```

## LIBRARY ROUTINES

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

- `upc_localsizeof(p)`: size of the local portion pointed by `p`
- `upc_blocksizeof(p)`: blocking factor associated with object pointed by `p`
- `upc_elemsizeof(p)`: size of the leftmost type of object pointed by `p`

## DYNAMIC MEMORY ALLOCATION

Three different memory allocation methods are provided by UPC:

- `upc_alloc(n)`: allocates at least  $n$  bytes of shared space with affinity to the calling thread; needs to be called by one thread only
- `upc_global_alloc(n, b)`: globally allocates  $n \times b$  bytes of shared data distributed across the threads with a block size of  $b$  bytes; needs to be called by one thread only
- `upc_all_alloc(n, b)`: collectively allocates  $n \times b$  bytes of shared data distributed across the threads with a block size of  $b$  bytes; needs to be called by all the threads
- `upc_free(p)`: frees shared memory pointed to by `p` from the heap

## STRING FUNCTIONS IN UPC

Equivalent of `memcpy`:

```
upc_memcpy(dst, src, size)
```

Copy from shared memory to shared memory.

```
upc_memput(dst, src, size)
```

Copy from private memory to shared memory.

```
upc_memget(dst, src, size)
```

Copy from shared memory to private memory.



## INDEX

---

- affinity, 7, 19, 20, 21, 23, 35, 49, 51–54, 60, 89, 125
  - `upc_forall` with `continue` as affinity argument, 53
  - `upc_forall` with expression as affinity argument, 53
  - `upc_forall` with integer as affinity argument, 52
  - `upc_forall` with shared address as affinity argument, 52
- ANSI C, *see* ISO C
- backtracking, 62
- barrier synchronization, *see* `upc_barrier`
- block size, *see* data layout
- blocking factor, *see* data layout
- bulk transfers, *see* string functions
- cache coherence, 117
- clusters, 116, 119
- collective function, 73
- `continue`, 53
- data decomposition, *see* domain decomposition
- data layout, 25, 51
  - \*layout qualifier, 28
  - three-dimensional cells, 60
  - blocked array, 25, 27, 33, 35, 37, 50
  - default data distribution, *see* round-robin data distribution
  - default block size, 86
  - indefinite block size, 27, 83, 86, 88
  - multidimensional array, 54, 60, 77, 84
  - round-robin fashion, 6, 50, 79
- deadlock, 105
- distributed memory multiprocessors, *see* DMs
- distributed shared memory, *see* DSM
- DMs, 116, 118
- domain decomposition, 49
- DSM, 18, 19, 116, 117
- dynamic memory allocation, 21
- embarrassingly parallel, 67
- examples
  - bakery algorithm, 110, 112, 113
  - dining philosophers, 106
  - heat conduction, 56, 58, 61, 62, 74, 80, 84, 93, 95, 129, 132
  - hello world, 2, 3
  - image histogramming, 101, 103
  - matrix–vector multiplication, 26, 29, 50, 52, 53, 81
  - matrix squaring, 43, 45, 46
  - N*-queens, 65, 68
  - parallel binary tree, 133
  - pointer arithmetic, 35, 37–39
  - pointer representation, 41
  - sparse matrix compression, 87
  - temperature converter, 4, 6–8, 10, 11, 13
  - vector addition, 25
  - wavelet transform, 90
- ghost zone, 54, 131
- ISO C, 1, 14, 21, 54, 73
- iterations, 59
- ITPACK format, 86
- locality, *see* affinity
- lock, 9
- `malloc()`, 82
- memory access order, *see* memory consistency
- memory allocation
  - deallocation, 73, 89
  - dynamically allocated, 73
  - statically allocated, 61
- memory consistency, 9, 108, 125
- message-passing interface, *see* MPI

- MPI, 19
- MYTHREAD, 3, 20, 49, 51
- nonblocking lock, *see*
  - upc\_lock\_attempt
- nonuniform memory accesses, *see*
  - NUMAs
- NP problem, 62
- NPB benchmarks, 61
- NUMAs, 117
- OpenMP, 19
- optimization, 115, 123
- parallel vector processors, *see* PVPs
- performance, 115
  - compilers, 122
  - optimization, 123
  - overhead, 120
  - run-time systems, 122
- pointers, 33
  - \*shared, 34
  - arithmetic, 35, 37, 39, 45
  - casting private to shared, 38
  - casting shared pointer to private, 38
  - casting shared to private, 43
  - casting shared to shared, 39
  - data representation, 37, 40, 42
  - handling functions, 42
  - private pointer to private, 14, 22, 33
  - private pointer to shared, 12, 22, 23, 34, 43, 46, 77, 79, 82, 83
  - shared pointer to lock, 104
  - shared pointer to private, 22, 34
  - shared pointer to shared, 14, 34, 82, 84
- pragma, 109, 112
- prefetching, 131
- private variables, 4, 21
- privatization, 124, 127, 129
- programming model, 17, 20
- PVPs, 116, 119
- race conditions, 9, 102
- relaxed, *see* upc\_relaxed.h
- remote access, 11, 19, 29
- remote accesses aggregation, 131
- shared arrays, 23
- shared variables, 3, 21
- SIMD, 117
- single program, multiple data model, *see* SPMD model
- SMP, 116, 117
- sparse matrix, 86
- split-phase barrier, 84, 98
- SPMD model, 2, 14, 19, 20, 50
- strict, *see* upc\_strict.h
- string functions, 127, 131
- struct, *see* structures
- structures, 23
- symmetric multiprocessors, *see* SMPs
- synchronization, 8, 91
- synchronization phase, 95
- system architecture, 116
- THREADS, 3, 20, 49, 51
  - dynamic number, 30
  - static number, 30
- trees, 62, 67
- UMA, 117
- uniform memory accesses, *see* UMAs
- upc\_addrfieldof, 40, 42
- upc\_all\_alloc, 73, 74, 77, 86, 89
- upc\_all\_lock\_alloc, 99, 102
- upc\_alloc, 82, 86, 89
- upc\_barrier, 8, 11, 59, 78, 92, 96, 99, 126
- upc\_fence, 110, 113, 126
- upc\_forall, 11, 25, 44, 49, 51–53, 59, 89, 103, 104, 125
  - affinity argument, *see* affinity
- upc\_free, 89
- upc\_global\_alloc, 78–80, 86, 89
- upc\_global\_lock\_alloc, 100, 104, 108
- upc\_local\_alloc, *see* upc\_alloc
- upc\_lock, 100, 105, 126
- upc\_lock\_attempt, 100, 106, 108
- upc\_lock\_free, 100
- upc\_lock\_t, 99, 126, 134
- upc\_memcpy, 127
- upc\_memget, 127
- upc\_memput, 127, 131
- upc\_notify, 94, 98, 99, 126
- upc\_phaseof, 40, 42
- upc\_relaxed.h, 108, 109, 111, 125
- upc\_resetphase, 142
- upc\_strict.h, 108, 109, 111, 125
- upc\_threadof, 40, 42, 52
- upc\_unlock, 100, 126
- upc\_wait, 95, 98, 99, 126
- von Neumann, 17
- work distribution, 49, 68